TU WIEN Informatics

Identifying Frameworks in Android Applications using Binary Code Function Similarity

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Yannik Zeier, BSc
Matrikelnummer 12202217

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dipl.-Ing. Dr. techn. Martina Lindorfer, BSc
Mitwirkung: Jakob Bleier, BSc BSc MSc

Wien, 29. Jänner 2024

Yannik Zeier

Martina Lindorfer

TU WIEN Informatics

Identifying Frameworks in Android Applications using Binary Code Function Similarity

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Yannik Zeier, BSc
Registration Number 12202217

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dipl.-Ing. Dr. techn. Martina Lindorfer, BSc
Assistance: Jakob Bleier, BSc BSc MSc

Vienna, January 29, 2024

Yannik Zeier

Martina Lindorfer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Yannik Zeier, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Jänner 2024

Yannik Zeier

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Yannik Zeier, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Yannik Zeier, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

Datum / Date: 29.01.2024

Unterschrift/Signature: 

Acknowledgements

I want to express my thanks to my thesis supervisors, Professor Martina Lindorfer and Jakob Bleier, for their guidance and support throughout my thesis. Their expertise and feedback have been invaluable.

I would also like to extend my deepest appreciation to my parents for their constant support and belief in my abilities.

Kurzfassung

Frameworks spielen oft eine zentrale Rolle bei der Entwicklung mobiler Anwendungen und steigern die Effizienz des Entwicklungsprozesses. Vor allem Cross-Platform Development Frameworks (CPDFs) haben aufgrund ihrer Fähigkeit die Entwicklung von Anwendungen für verschiedene Plattformen durch Verwendung einer gemeinsamen Codebasis zu beschleunigen, an Beliebtheit gewonnen. Trotz der weiten Verbreitung dieser Frameworks gibt es nur wenig Forschung, die sich auf ihre Erkennung in Anwendungen konzentriert, zudem gibt es keine Vergleiche zwischen den verfügbaren Detektoren. Die vorliegende Thesis schließt diese Forschungslücke, indem sie eine Liste aktueller Framework-Detektoren vorstellt, ihre Funktionsmechanismen untersucht und eine Vergleichsanalyse durchführt. Anhand eines Datensatzes mit 524 Android-Anwendungen, die mit verschiedenen Frameworks entwickelt wurden, setzen wir verschiedene Obfuscation-Techniken ein, um deren Auswirkungen auf die Erkennung der verschiedenen Detektoren zu bewerten. Unsere Untersuchung zeigt, dass konventionelle Detektoren durch einfaches Umbenennen von Klassen, Dateien, Verzeichnissen und Bibliotheken, mithilfe simpler Bash-Skripten, umgangen werden können. Motiviert durch diese Erkenntnisse stellen wir einen neuartigen Framework-Detektor vor, der mithilfe einer Binary Code Similarity Metrik Funktionsmerkmale vergleicht, welche aus der binären Of Ahead Time (OAT) Repräsentation der Anwendungen extrahiert wurden. Die Ergebnisse unserer Analyse zeigen das Potenzial der, aus den binären OAT Repräsentation extrahierten Funktionsmerkmalen, für die Framework-Erkennung auf und motivieren weitere Forschung in diesem Bereich.

Abstract

Frameworks play a pivotal role in expediting the development of mobile applications, enhancing efficiency of the development process. Notably, Cross-Platform Development Frameworks (CPDFs) have garnered significant attention for their capability to facilitate the creation of applications across various platforms through a unified codebase. Despite the widespread adoption of these frameworks, there exists little research focused on detecting their usage in applications, with no comprehensive comparisons among available detectors. This thesis addresses this research gap by presenting a list of current framework detectors, delving into their operational mechanisms, and conducting a comparative analysis. Leveraging a dataset comprising 524 Android applications developed using known, diverse frameworks, we employ various obfuscation techniques to assess their impact on the performance of different detectors. Notably, our investigation reveals that conventional detectors can be thwarted by simple renaming techniques, such as class, file, directory, and library renaming, applied using simple bash scripts. Motivated by these findings, we introduce a novel framework detector that relies on binary code similarity metrics comparing function features extracted from the applications' binary Of Ahead Time (OAT) representation. The results of our analysis underscore the potential of utilizing function features derived from binary OAT representations for framework detection, highlighting a promising avenue for further research in this domain.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Developing applications is a time-consuming and costly task, and with users on different platforms, e.g. Android, iOS, and Windows, that each require additional development effort, the number of applications utilizing frameworks is rising, with a recent study of Biørn-Hansen et al. [BHGM+22] finding that 15% of the 661,705 analyzed apps on the Google Play Store were build using a framework. These so-called Cross-Platform Development Frameworks (CPDFs) allow developing applications using pre-made templates, components and structures, whilst abstracting away from many details allowing to reuse the same code for different platforms, hence requiring fewer developers with platform specific knowledge and reducing development time.
Knowing which framework is used in an application is a vital first step in deciding how to further process and analyze the application, as it has impact on the way the application's code is structured and executed, as well as the programming language it is written in. Especially the latter heavily impacts the decision on which tools and analysis techniques can and should be used in order to achieve the best possible results. Therefore, a reliable method of detecting the framework in an Android application given its APK file is needed. Such a detector is not only useful in the realms of application analysis e.g. for security research, but can also be used a basis for various other research, where the information about the frameworks used is of interest, e.g. investigating trends and patterns in program development.

Although several papers and tools rely on simple string, class name and file presence checks for detecting the framework used in an Android app, we create a novel technique utilizing binary code analysis on the application's Of Ahead Time (OAT) binary representation, an approach introduced by Bleier and Lindorfer [BL23]. Section 2.1.3 will explain the OAT representation in more detail. As we show in chapter 4 approaches used by the current framework detectors can be easily circumvented, hence, there is a need for

an improved and more robust technique for framework detection.
Android applications are often protected using obfuscation, in order to protect their intellectual property from being stolen, as well as to hinder decompilation and reverse engineering [WHA+18, DLD+18]. These obfuscations complicate application analysis, requiring robust analysis tools, in order to remain effective [WWZR18]. Hence, we examined different current framework detection techniques and evaluated their effectiveness against obfuscated as well as non-obfuscated Android applications. To the best of our knowledge there exists no research focusing solely on framework detection. This is surprising as several papers utilize framework detection for achieving their research objectives. As a result, there is no research comparing and evaluating existing framework detection techniques.

1.2 Contributions

Summarised, the main contributions of this thesis are:

- Providing an overview over existing framework detection tools and their functioning.
- Creation of a novel approach for detecting the framework used by an Android application, based on the application's OAT representation.
- A comparative study of our framework detection tool and existing tools on a dataset of 524 applications.
- Comparison of the obfuscation resilience of different tools by applying various types of obfuscation to the evaluation dataset.
- An analysis and evaluation of the results discussing strengths and weaknesses of the different techniques and discovering possible future improvements.

1.3 Research Questions

This thesis we will aim to answer the following research questions.

R1 - How do the current framework detection tools perform compared to each other?

R2 - How does obfuscation affect the performance of the different framework detection tools?

R3 - How does our novel approach perform compared to the tools tested in **R1** and **R2**?

R4 - Is binary code analysis of an Android applications OAT representation a feasible approach for detecting its used frameworks?

1.4 Structure of the Thesis

The remainder of the thesis is structured as follows. In chapter 2, we will take a look at background information needed for understanding this work. This includes information about the structure of Android Packages (APKs) and Android App Bundles (AABs) files, as well as the building process for Android applications (Section 2.1). Furthermore, we explain CPDF (2.2) and common obfuscation techniques used with Android applications (Section 2.3), as well as binary code similarity and their general functioning (Section 2.4).
In chapter 3, we will take a look at existing framework detection techniques, the way they work, and their downsides, before discussing the current state of library detection. Section 3.3 lists some key research results and techniques for binary code similarity.
We show the current framework detection techniques to be susceptible to common obfuscation techniques in chapter 4, by providing proof of concept scripts and command against several detectors.
The framework detector we developed as part of this work is introduced and explained in chapter 5, together with the dataset used for creating and evaluating our detector.
Chapter 6 contains the evaluation of the different detectors and discusses its results.
The thesis concludes with chapter 7 summarising key findings and formulating possible future work.

CHAPTER 2

Background

This section covers fundamental information needed for understanding the topic of this thesis. In section 2.1, we explain the structure of APKs, their function as well as contents and how they differ from AABs. Furthermore, we will look at the build process of native Android applications. Section 2.2 will explain frameworks, define different framework categories and explain their advantages and disadvantages. In addition, we will have a look at two frameworks to see how they work. In section 2.3, we will look at different obfuscation techniques. Lastly, section 2.4 introduces fundamental knowledge about binary code similarity.

2.1 Android Basics

2.1.1 Android Package Structure

An Android Package (APK) is a signed archive file representing the application and can be installed on the device. It contains the application's contents required at runtime. We will list and explain them in the following [Andf, Andd, Andk, Ande, ZBLO21, DLD+18, LB22].

AndroidManifest.xml is the most important file, as it contains essential information required for running the application. This includes the application's name and icon, the permissions it uses, hardware requirements as well as the minimum Android version supported. It also declares the app's components, i.e. activities, services, content providers and more. In addition, it defines how other applications are allowed to interact with the application.

META-INF/ is directory containing metadata files like the signatures file, the *MANIFEST.MF* file, which lists all files in the application package, as well as the *CERT.SF* file containing cryptographic hashes of the files mentioned in the manifest [Andk]. These hashes are used to ensure the integrity of the application.

resources.arsc contains resources that can be compiled together in an XML file, e.g. strings. Additionally, it contains paths to the content not included in this file, such as images and layout files.

res/ is the directory containing the resources, that were not compiled into the *resources.arsc* file, e.g. icons, layout, fonts.

assets/. This directory contains the app's raw asset files. It is similar to the *res/* directory but gives the developer more freedom in how to structure the data located here, i.e. creating subdirectories and arbitrary file structures are allowed. Files saved in the *assets/* directory are not given a resource ID and, therefore, have to be accessed using the *AssetManager* [Ande].

lib/ contains platform dependent compiled code, like native libraries. As each CPU architecture requires a different version of the native libraries, this directory is split into subdirectories, e.g. arm64-v8a, x86, x86_64, each containing the libraries compiled for that particular architecture.

classes.dex contains the compiled classes in Dalvik Executable (DEX) file format. This is the executable run on the device. As one DEX file can at most reference 65,536 methods, it is possible for multiple DEX files to be present. In that case their naming follows the scheme of *classes1.dex*, *classes2.dex*... [Andc].

2.1.2 Android App Bundle

Since August 2021, publishing Android applications on the Google Play Store, the largest market for downloading Android apps [Stab], requires the developer to publish their applications as an Android App Bundle (AAB) [Andb]. An Android App Bundle is an archive file that, in contrary to APK files, can not directly be installed on a device [Andf]. It contains the contents of an Android app project and additional metadata that can be used to generate and sign the APK file later. This allows Google Play's servers to generate APKs optimized for the particular device requesting to install an application, by for example removing unneeded code and resources, e.g. x86 native libraries. We mentioned App Bundles for completeness, but they will not have any further relevance for this work. It is to be noted though, that a side effect of using AABs is that there can exist multiple APKs for the same version and code of an application that have different hash values.

2.1.3 Building Process

Note that if not state otherwise, we refer to the building process of native Android applications, i.e. applications not built using some framework. The building process when using frameworks can differ depending on the used framework. We will see some examples for building applications using frameworks in section 2.2. Earlier, we mentioned AABs, these are generated differently. However, the APK installed on the user's device will be generated the same way as described below using the AAB [Andg].

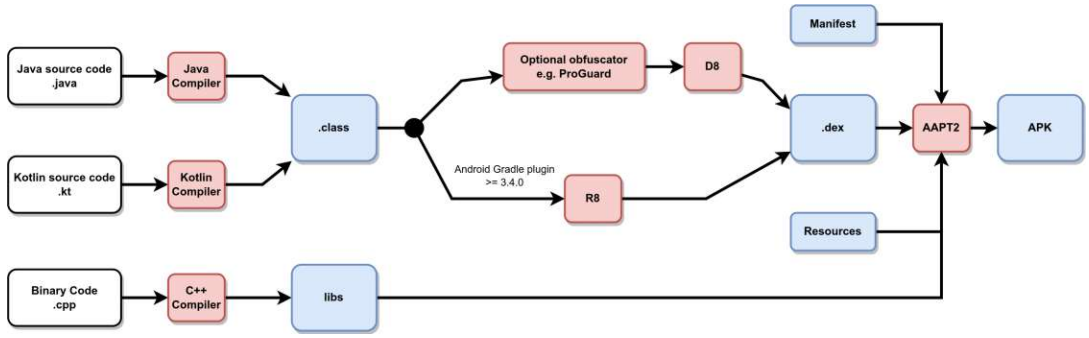

Figure 2.1: Android Build Procedure

Android applications can be written using Java, Kotlin and C/C++. Both Java and Kotlin can be used interchangeably for developing the actual main part of the application [Andf]. Whilst C/C++ is used for developing libraries using the Java Native Interface (JNI) for interacting with the application [Andj]. As we will see in section 2.3 using native libraries is a possible way of obfuscating an app's behavior.

Figure 2.1 shows the general compilation procedure of an Android application. Depending on whether the application is written in Java or Kotlin the source code is compiled to *.class* files with the Java or Kotlin compiler [Andh, Now]. From there on, the steps are the same for both Java and Kotlin applications. The generated *.class* files can now be obfuscated using, for example ProGuard or any other obfuscator before being converted to Dalvik byte code.
Before version 3.4.0 of the *Android Gradle plugin* released in April of 2019, obfuscation was done using ProGuard, after which the D8 compiler was used to convert the Java bytecode to Dalvik bytecode. When using newer versions of the Android Gradle plugin for building applications, only the R8 compiler is used. This is an optimized version of the D8 compiler that can also handle obfuscation [Andl, Med, Andi, Lev]. Note that ProGuard and R8 are part of the official Android Build chain and, therefore, used by many applications. The obfuscation performed by *R8* is optional and disabled by default. In addition, there exist plenty commercial third-party obfuscation tools, like DexGuard [Dexa], DashO [Das] and DexProtector [Dexb], that can be used at different parts of the build procedure.

The resulting *.dex* files will then be packaged together with the app's resources, the manifest file and the used libraries into an APK using theAndroid Asset Packaging Tool (AAPT2) [Anda]. The generated APK file can then be installed and run on the device. Android versions prior to 4.4 run applications on the Dalvik Virtual Machine, whilst newer versions use the Android Runtime (ART) [Med, Lev, LB22]. The Dalvik Virtual Machine is an adaption of the Java Virtual Machine (JVM) optimized for more restricted lower power devices. The main difference between the Dalvik VM and the ART is the way they handle code compilation. Both interpret the Dalvik bytecode given to them, this however is rather slow. Therefore, they also make use of Just-in-Time (JIT) compilation.

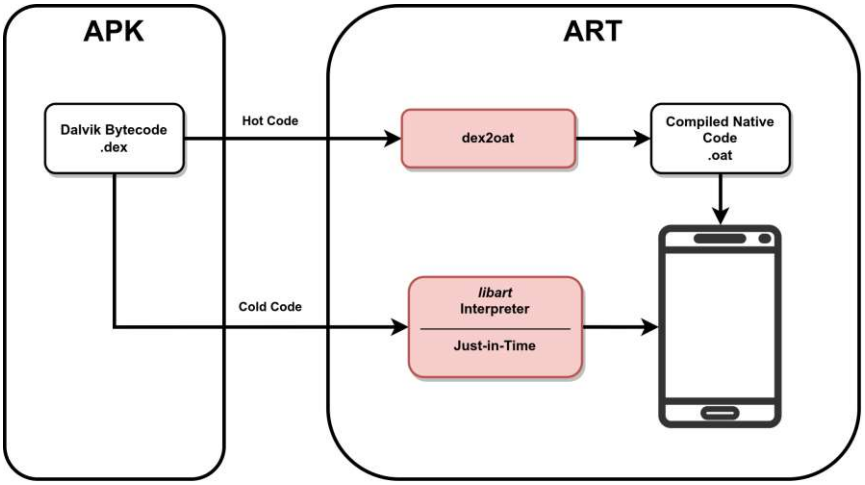

Figure 2.2: Launching an application using the Android Runtime

Here the systems compiles often and repeatedly used code, so-called hot code, into binary code. This increases performance as the compiled code is usually faster and better optimized, furthermore, the system can execute it directly on the CPU without first having to interpret it.
The ART goes one step further and also makes use of Ahead-of-Time (AOT) compilation. Here the systems remembers often used code parts across application launches and compiles these into binary code, storing them as *.oat* files, which can be loaded from disk upon program launch. An OAT file is an ELF shared object that contains additional sections with OAT metadata [lie]. Figure 2.2 shows an overview of the application loading and executing process for the Android Runtime. Furthermore, it is possible to share information about which parts of an application are executed most between users. This can be done via the Play Store, resulting in the system being able to compile parts of the application upon installation or updating with the effect of having an optimized application on the first startup already [Lev].
In general, applications built using a framework also have to follow the structure explained in this section at some point in order to generate an APK or AAB file.

2.2 Frameworks

We will now have a look at frameworks. These are tools that allow developing applications using common abstractions, such as pre-made templates, components and structures. In contrast to developing apps natively for the Android system, the use of frameworks often abstracts away from many details allowing for reusing the same code for different platforms, e.g. iOS. There are also frameworks used for game development, e.g. Unity [Uni], however, these are not the main focus of this section, which lies on CPDFs. Hence, if not state otherwise, when talking about frameworks we refer to CPDFs for the following of this section. Popular examples are, Flutter [Flu] and React Native [Reab].

We will now look at some of the advantages of frameworks and why they are used by developers. One of the main reasons for using CPDFs is their ability to create applications for multiple platforms, with less effort required than to develop a native application for each platform. This reduces the development cost and effort. Furthermore, it can speed up the development as well, as only one application has to be developed and maintained. In addition, the developers only require knowledge about the framework itself and not specific knowledge about the various platforms the application will eventually run on. Framework applications are often written in programming languages different from the ones used for developing native applications. This allows developers new to a platform to quickly work on projects without having to learn the specifics for that platform first, especially as many frameworks use Web technologies, like HTML, CSS and JavaScript.

Due to the additional abstraction layers they tend to have worse performance and higher memory consumption than native applications [MA21]. The application size especially after installation is larger than these of native Android applications. A Study by Mahendra et al. showed that depending on the framework used, the application can be 10 times larger than a native one [MA21].
As CPDFs often wrap calls to the platforms' native APIs, in order to allow platform independent usage from within the framework, one can not easily access native APIs that are not supported by the framework. Some frameworks allow manually adding access to platform specific features, with the downside of reducing portability of the application in the process [MDC+21]. In general what is gained by the convenience and development speed of CPDFs is often paid for in reduced control over the own application and more limited access to native features.

In the following we will now look at 5 different categories of CPDFs. These categories were introduced by Xanthopoulos et al. [XX13] and further refined by Biørn-Hansen et al. [BHGG18].

- **Web apps** are browser based applications that render a Web page. This means the actual application part is a website built using Web languages like JavaScript and HTML, which then is rendered by a browser. The advantages lie in their platform independence, furthermore they do not require downloading or updating in the sense a conventional application would. Downsides are slower performance, requirement for an Internet connection, as well as the limited access to the platform the application is running on, as it has to rely on the APIs provided by the browser. Some frameworks for creating Web apps are Angular [Ang] and React.js [Reaa]. These type of frameworks are not important for our work, as we focus on frameworks producing an APK file.

- **Hybrid apps** are developed using Web technologies, like Web apps are. However, they are not run through a browser, but instead run inside a thin container application. This is a native application containing an element responsible for rendering the application Web code, which can be either a UIWebView on iOS or a WebView element on Android. This container can be thought of as an application

that just renders one particular website. Due to it being a native application it has access to the APIs of the particular platform it is running on, which allows for a better and deeper integration into the platform. However, they can not make use of platform optimized UI elements and have to rely on Web elements, therefore, sacrificing some performance. Examples for frameworks of this category are Apache Cordova [Apaa], Capacitor [Cap] and Framework7 [Fra].

- **Interpreted apps** make use of an interpreter to run framework specific code for each supported platform. They consist of a native part that is specific for each platform and the platform independent application part. Due to the interpreter being native to each platform it has access to its APIs and performance features, which it wraps in a such way that the application can access it independent of the actual platform. The developer therefore only has to know the APIs provided by the framework and does not need to learn the specific APIs of the various target platforms. The interpreter using platform specific features and elements leads to a user experience more similar to conventional applications, due to the fact that native UI elements can be used to display content instead of relying on HTML elements with CSS styling. Often times the application code itself is still written in Web Languages similar to the hybrid approach, but instead of being executed in a WebView they are interpreted by an interpreter like JavaScriptCore or V8. Examples for frameworks of this category are React Native [Reab], NativeScript [Nat] and Titanium SKD [Tit].

- **Generated apps** sometimes also called cross-compiled apps are written for one platform, and get converted and compiled like a native application for all other supported platforms. They achieve very good performance as they are basically native applications. However, the automatically generated code from the conversion step can be quite complex and hard to debug as the conversion step is non-trivial. Popular examples are Flutter [Flu], which uses Dart for the application code, and Xamarin [Xam], which uses C#.

- **Model-Driven** frameworks use the paradigm of Model-Driven Software Development. Here the native application code is generated and derived from domain specific language that can be learned and used by developers and non-developers alike. This language is used to describe the application and its behavior. In most cases this language is specific to the framework used.
  MD² [MD2] is an example for a framework of this type. For practical use cases model driven frameworks are rather uncommon, they are more prevalent in research [BHGG+19, BHGG18].

In the following we take a look at the functioning of React Native and Apache Cordova, as these two frameworks are commonly used and influential, with Apache Cordova being used as the basis for many other frameworks, such as Ionic, Capacitor and Framework7 [BHGG+19, MS21, Ionb, BHGG18].

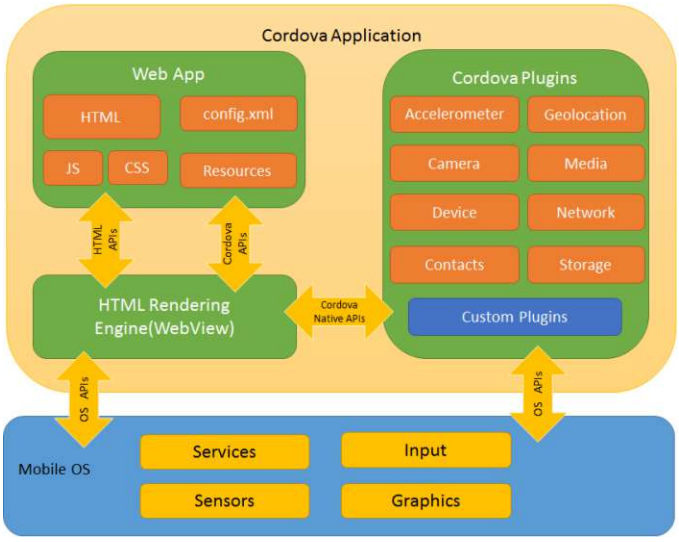

Figure 2.3: Architecture overview of Apache Cordova

Taken from `https://cordova.apache.org/static/img/guide/cordovaapparchitecture.png` at 2023-01-27

React Native [Reab] is an interpreted CPDF [Read]. This means that it has a native component that will interpret and run the framework specific code, which in the case of React Native is written in JavaScript. The component translating between these two, is called the React Native bridge. It sits between the application logic written in React Native specific code and the native part of React Native responsible for rendering the UI and interacting with the actual platform. Events on the native site will get translated by the bridge and send to the application logic, where they are handled [Reac, Coo].
With version 0.68 React Native changed its architecture, such that native parts use the JavaScript Interface (JSI) for communication between the native and the JavaScript code. This optimizes the communication and thus increases performance [Read, Reae, Pat].
When generating an Android APK using React Native the resulting APK still follows the structure shown in section 2.1.1. The application code is added as an asset and gets loaded by the native part for interpretation upon launch.

Apache Cordova [Apaa], formerly named PhoneGap [Phoa, Phoc, Phob] is a cross-platform development framework using the hybrid approach. It allows developing for Android, iOS and Electron using standard Web technologies, i.e. HTML, CSS and JavaScript. Figure 2.3 gives an overview of Apache Cordova's architecture. The actual application is executed inside a wrapper, e.g. Android WebView, handling the HTML rendering and JavaScript execution. This wrapper is run in the native part of the application that also contains plugins accessing platform specific features, e.g. location information or the camera. Access to these is provided to the application via the wrapper component.

2.3 Obfuscation techniques

There are many different obfuscation techniques applicable to Android applications. The reasons a developer might want to obfuscate his application's code can be diverse. Many Android applications use obfuscation in order to protect its code against theft of intellectual property or so called repackaging attacks [LBK21, LLB+17, WHA+18, DLD+18]. In a repackaging attack a malicious actor takes an application and modifies something before re-releasing it. This can be done in order to spread malware or redirect advertisement revenue to the attacker [WHA+18, LLB+17, LBK21]. Repackaging attacks are one of the major concerns mobile developers have when it comes to protecting their code [LBK21]. But obfuscation can also be used to hide malicious functionality and thereby, is also often used by malware [DLD+18].

In the following we describe some common obfuscation techniques used with Android applications [ZFL+20, LLJG15, BBM+18, ZBLO21, DLD+18, MdTGM19]. Many of them, like dead code removal and identifier renaming, can be enabled in the default Android build chain [Andl].

1. **Package Flattening**, here the package hierarchy is changed by moving files to other directories or removing subdirectories all together. This prevents using the file hierarchy to derive the application's structure, i.e. code belonging to the same library [ZFL+20, LLJG15].

2. **Identifier renaming**. Identifiers like field, method, and class names carry a lot of information that may be useful for analysis or reverse engineering. By renaming these to random strings this information is lost.

3. **String Encryption**. Here sensitive strings in the code are encrypted such that they are meaningless and can not easily be extracted. Together with reflection it can hide, which classes or methods are actually used by a reflection call.

4. **Dead Code Removal**, is used to remove unnecessary code parts. This can lead to features extracted from the code to change, whilst its functionality remains unchanged. It is often used as an optimization in order to reduce an application's size.

5. **Control Flow Modification**, here the control flow is changed without altering the executed tasks. This can be done in various ways, the results however are always the same and aim at disrupting tools and techniques using the control flow for analysis or identification. An example for such a technique, is control flow flattening [LK09]. Here every jump instruction between two basic blocks is made via a dispatcher node that jumps to the actual location based on a variable set at the end of every block. The dispatcher block can be thought of like a switch statement.

6. **DEX Encryption** allows developers to encrypt the whole DEX file. This prevents reverse-engineering tools from directly accessing the functions and components [ZFL+20].

7. **Junk Code Insertion**. Analysis techniques that use features, signatures comprised of code snippets or statistics of the code can be manipulated by inserting junk code, which will not be used by the application. Detecting and ignoring such unused pieces of code is rather easy, and as a result this technique is not as effective as others [BBM+18, MdTGM19].

8. **Native Code**. The DEX code of Android applications is usually easier to analyze, than x86 code for example, as it contains more information [DLD+18, ZBLO21]. Outsourcing code into native libraries can it make harder to analyze them, as these contain less information and require further analysis tools.

9. **Reflection**. This is a feature of Java and Kotlin that allows for inspecting, loading and interacting with classes, files and methods at runtime. It can therefore be used to alter the programs execution and structure during runtime making static analysis less concise.

10. **Asset/Resource Encryption**. Here the assets and resources are encrypted. Access to them is wrapped with a decryption routine. Often the assets and resources are also randomly renamed to prevent filenames from disclosing their content or origin.

11. **Library Encryption**. The libraries used by an application can tell a lot about it and the operations it performs, hence library detection has been studied quite extensively [ZLF+22, ZFL+20, MWGC16, LLJG15, WWZR18, BBD16]. During library encryption the library files itself can be encrypted, altered and renamed to prevent identifying them. Similar to the asset and resource encryption calls these libraries are wrapped with a routine, that decrypts the libraries before loading them during runtime if needed.

A study by Wermke et al. [WHA+18] found that 25% of the analyzed applications make use of some form of obfuscation. This number raises to around 50% when only looking at applications with more than 10 million downloads. This shows that there is a non-negligible number of Android applications using obfuscation. Analysis tools not able to handle obfuscated apps are therefore at a clear disadvantage. Furthermore, third-party libraries are more likely to be obfuscated. As the majority of Android applications use third-party libraries [LLJG15, Exo, ZLF+22], it is even more necessary for analysis tools to function with obfuscated code.

2.4 Binary Code Similarity

The approach of compiling the APK-files into a native binary format allows the usage of code binary similarity techniques, typically not applicable for Android applications. Binary code refers to machine code that can directly be run by a CPU and is typically produced by a compilation process from a higher level programming language. Binary code similarity techniques are used for comparing compiled programs without the use of their source code. Comparison can be done at different granularity levels, e.g. instructions, basic blocks, functions or whole programs.
It is used for many different tasks like bug search, malware clustering, malware detection, malware lineage, patch generation, patch analysis, porting information across program versions and software theft protection [HC21, LHZ+18].

To be considered similar two code pieces have to be similar in their syntax, structure or semantics. Syntax similarity compares the binary code's syntax, i.e. the concrete instructions in the code. On the other hand, semantic similarity does not rely on the syntax representation of code, as it focuses on the meaning and functionality of the code. Two programs can be syntactically completely different but semantically the same. A simple example are the statements `a++` and `a += 1`. They both perform the same task of incrementing the variable `a` by 1 whilst using a different syntax. Techniques using structural similarity lay in between these two and work on graph features extracted from the source code, i.e. Control Flow Graphs (CFGs) and Call Graphs (CGs). These graphs contain part of the program's semantics, whilst at the same time being influenced by its syntax. For example, by inlining a function, the program's CG changes without the semantics being changed. Nevertheless, graph based techniques are often used due to entirely semantic approaches being computationally more expensive.
Relying on syntax similarity is not robust like semantic similarity as the same source code compiled at different times, with different compiler settings or a different compiler can lead to different binary code [HC21]. However, as some features and structure persists, using structural features, allows for matching two code pieces without the need of complex semantic based approaches or uninterpretable machine learning techniques, as has been shown by Kim et al. [KKC+23]. The problem with most binary code similarity techniques using machine learning for identifying and extracting features is the fact that they are hard to interpret and debug. This is due to the complex and abstract nature of the features learned by the machine learning models, making it challenging for human analysts to understand the underlying patterns. This makes improvement and adaption of the techniques harder.
In general, binary code similarity is performed in four major steps, described by Kim et al. [KKC+23] as below:

1. *Syntactic Analysis.* In this first step an intermediate representation like the disassembled machine code or an Abstract Syntax Tree (AST) is created from the provided binary code. This is the basis all following steps are build upon.

2. *Structural Analysis.* During this step control structures, like the CFG and CG are created from the intermediate representations created in the previous step.

3. *Semantic Analysis.* In this step additional semantic information is inferred from the binary code to capture additional program semantics and enhance the structures and information recovered in the first two steps. This can for example be done using data flow analysis.

4. *Vectorisation and Comparison.* In this final step, the information recovered in the previous three steps is vectorised in order to allow for computing a similarity score.

Depending on the actual technique and features some the first 3 steps can be skipped. Examples for concrete techniques can be found in section 3.3.

CHAPTER 3

Related Work

Detection of frameworks on Android has not been in the focus of scientific research. To the best of our knowledge there are not any publications about framework detection specifically. However, it has been used as a tool in some studies dealing with CPDFs. Library detection, on the other hand has already been studied to a great extent. Detecting and identifying the libraries used in an application faces similar challenges as detecting the used framework. We will mention previous work on library detection (Section 3.2) as it can be used to identify the framework used in an app, even though we have not seen any detectors use libraries apart from their filenames for detection. Lastly section 3.3 will give an overview about existing research and techniques for determining the similarity of binary code, as the framework detection technique introduced in section 5.3 of this thesis builds upon techniques and findings of previous research on binary code similarity.

3.1 Framework Detection

This section will summarize several previous research papers, that made use of framework detection as part of their work. After that we have a look at tools capable of detecting the frameworks used in an application. Table 3.1 on the end of this section contains an overview over the different techniques and the frameworks they detect.

The first large scale analysis detecting frameworks in Android apps was done by Viennot et al. [VGN14] in 2014 as part of their *PlayDrone* application for scraping and analysing apps from the Google Play Store. Their approach works by looking at the directory hierarchy of the decompiled APKs searching for the fully qualified names of 3 frameworks. Hence, it is not robust against hierarchy flattening and directory renaming obfuscation as introduced in section 2.3. It can detect PhoneGap (now called Apache Cordova), Adobe Air and Titanium.

Another early work on detecting frameworks in Android apps was done by Malavolta et al. [MRST15] for investigating the usage of cross-platform apps on the Play Store, i.e. rating, review count and used libraries. Their technique searches the decoded APK for specific file names and file extensions, as well as specific strings, in order to determine the framework used by the app. The tool itself is available on GitHub[1], but has not been updated since 9 years, together with the fact that the majority of frameworks it claims to detect have been deprecated and are no longer being developed the practical use of this tool is limited.

Ali and Meshba [AM16] claim to have publicised the first study investigating cross-platform applications on the Play Store. However, this is not the case as both Malavolta et al. [MRST15] and Viennot et al. [VGN14] published prior research. For framework identification their tool extracted the list of class names using *android-classyshark*[2] and searched the list for the fully qualified class names of three frameworks: PhoneGap/ApacheCordova - "org.apache.cordova", Appcelerator titanium - "org.appcelerator.titanium", Adobe Air - "com.adobe.air". Their claim to identify cross-platform applications with an 100% accuracy has to be questioned as obfuscating the class names would easily prevent detection as we show in chapter 4.

Mohanty and Sridhar [MS21] extend the approach of Ali and Mesbah [AM16] to detect more Apache Cordova based frameworks, like Ionic, Monaca, OnsenUI, Phonegap and Framework7, but also to detect apps using React Native. The main contribution of their paper is the identification and investigation of security issues in applications using cross-platform frameworks. Their tool, called *HybriDiagnostics*, searches the unpacked APK directory for the filename *cordova.js* to infer that the app makes use of an Apache Cordova based framework. In addition, the *classes.dex* file is disassembled into a human-readable format using *dexdump* and searched for *<Class-Descriptor>* strings in order to extract the fully-qualified class names of all compiled classes. The list of extracted class names is then searched for class names associated with the specific frameworks. The concrete class names searched for, as well as *HybriDiagnostics* itself, have not been made public by the authors.

Another approach inspired by Ali and Meshba [AM16] was developed by Biørn-Hansen et al. [BHGM+22] and used for investigating the usage of different CPDFs on the Google Play Store over time. Their technique works by searching for certain strings in the *AndroidManifest.xml* and by searching for certain files, file extensions and folders in the */assets/* directory and the APK file. With supporting 13 different up-to-date frameworks it is the technique capable of detecting the most distinct frameworks we have found. Their program code as well as the dataset created with it has been published on GitHub[3].

The framework detection tools looked at so far have been created as part of other research. We could not find any commercial tools advertising the capability of detecting

[1] https://github.com/GabMar/ApkCategoryChecker
[2] https://github.com/google/android-classyshark
[3] https://github.com/andreasbhansen/phd-thesis-contributions

an application's framework. There are however some publicly available tools claiming to be capable of detecting the framework used in an Android application. In addition, these are the most up-to-date techniques and tools available.
One of them is the APK Framework Detector tool [APKa]. It works by unpacking the APK and looking for known filenames and their locations. It has a static list of known files for different frameworks, mostly made up of framework specific library files.
Another tool also searching for specific filenames in order to determine the framework used by an application is Mob Framework Radar [Mob]. Even though these techniques are simple and fast they are not robust against obfuscation, as simply renaming files will lead to them failing. Furthermore, one could easily add files with the corresponding names to trick these tools into detecting a wrong framework.
DroidLysis [Dro] is a tool for pre-analysis of Android apps, however, it has a feature for detecting third party libraries by searching for certain strings the directory hierarchy. As a part of this feature it allows detection of some frameworks and can therefore also be considered to be a framework detection technique.

Two closed source tools for framework detection distributed as Android apps are PowerAPK [Pow] and APK Platform Detector [APKb].
PowerAPK also looks for the file names of used native libraries and other known filenames, e.g. */assets/www/cordova.js*. According to a blog post of the tool's author [Sis], it also checks for known substrings in these files. We also confirmed this by reverse engineering and analysing the APK downloaded from the Google Play Store using jadx[4].
APK Platform Detector [APKb] is another tool working similarly. Using jadx to analyze the tool's APK we found that it also checks the package names in the application against a list of known package names used by frameworks.

All the existing tools for framework detection function by similar principles, which are vulnerable to simple obfuscation as shown in chapter 4. Furthermore, most techniques are only able to detect a subset of frameworks. To the best of our knowledge there exists no study comparing different framework detection techniques or investigating the impact of obfuscation to their effectiveness.

3.2 Library Detection

Library detection is related to framework detection. The first step of most library detection techniques is to find possible library candidates in the applications' code before identifying them. The similarities to framework detection lie in the part of identifying the code pieces and attributing them to a framework or library.

Zhan et al. [ZLF+22] conducted a literature review on the research of Third-Party Librarys (TPLs) on Android. They proposed a taxonomy for categorizing TPL research on Android systems by their research objectives, targeted libraries, types of TPLs and types of program analysis used. For each research objective they gave an overview about the

[4] https://github.com/skylot/jadx

Technique	Supported Frameworks		
Viennot et al. [VGN14]	• Adobe Air	• PhoneGap	• Titanium
Malavolta et al. [MRST15]	• Apache Cordova • Enyo • IBM Worklight • IUI	• Kivy • mobl • MoSync • Next	• QuickConnect • Rho Mobile • Sencha • Titanium
Ali and Meshba [AM16]	• Adobe Air	• PhoneGap	• Titanium
Mohanty and Sridhar [MS21]	• Apache Cordova • Framework7 • Ionic	• Monaca • OnsenUI • Phonegap	• React Native
Biørn-Hansen et al. [BHGM+22]	• Adobe Air • Apache Cordova • Capacitor • Codename One	• Flutter • Ionic • NativeScript • Qt Mobile	• React Native • Titanium • Weex • Xamarin
APK Framework Detector [APKa]	• Apache Cordova • Flutter	• React Native • Xamarin	
Mob Framework Radar [Mob]	• Apache Cordova • Capacitor • Expo	• Flutter • React Native • Uno Platform	• Xamarin
APK Platform Detector [APKb]	• Flutter	• React Native	• Xamarin
Power APK [Pow]	• Apache Cordova • Godot	• Mono • React Native	• Unity
DroidLysis [Dro]	• Apache Cordova • Flutter	• React Native • NativeScript	• Titanium SDK • Unity

Table 3.1: Framework detection techniques and the frameworks supported by them.

current state and existing research, as well as identified shortcomings and possible future research directions.
The research objectives investigated were, TPL detection, security issue analysis, TPL privilege de-escalation, TPL maintenance and TPL attribute understanding. TPL detection is the most relevant for our work.
All library detection tools investigated by Zhan et al. [ZLF+22] used static analysis techniques. Static analysis techniques are easier to scale and get better code coverage [CGO15]. Whilst dynamic techniques have the advantage of analysing the code that is actually being run, which makes them more robust against reflection and dynamic code loading. However, they are limited to analyzing code executed at runtime and, therefore do not analyze the complete application. Dynamic techniques are usually more suited for detecting malicious libraries, as these tend to more often make use of dynamic code loading and reflection, in order to hide their malicious activities [LLB+17].

In a previous paper Zhan et al. [ZFL+20] compared different library detection techniques in terms of their practical usage and implementation performance. They also listed the features and methods used for detecting and identifying possible library candidates.

Techniques mentioned and compared by the previous two papers include LibRadar [MWGC16, Libb], ORLIS [WWZR18], LibScout [BBD16, Libc] and LibPecker [ZDZ+18, Liba]. In general library detection is done in four steps [ZLF+22]:

1. Pre-Processing
2. Library Instance Construction
3. Feature Extraction
4. Library Instance Identification

During the pre-processing step the application is disassembled and transformed into a representation suitable for further analysis. Library instance construction refers to the identification of possible libraries, this means finding the pieces that make up each library. During the feature extraction, the features used for identification of the TPLs are extracted, before they are used in the last step to identify the library instances.

LibRadar [MWGC16] uses the package hierarchy to determine possible libraries, making it susceptible to obfuscation by package flattening and error-prone due to the fact that libraries might share the same root packages [ZLF+22]. It uses the frequency of Android API calls as features to cluster and identify the library candidates, making it more efficient in terms of computation resources but also more susceptible to obfuscation like dead code removal.

ORLIS [WWZR18] uses fuzzy method signatures as feature for identifying library candidates. For coming up with possible library candidates it uses class dependency relations extracted from the call graph. As these do not rely on the package structure

it is to best of our knowledge the only tool for library detection that is robust against package flattening. However, the quality of the extracted class dependencies can be impacted by modification to the control flow.

LibScout [BBM+18] claims to be able to detect specific versions of a library. For identifying a TPL it uses fuzzy method signatures, like ORLIS. The package hierarchy is used for constructing library candidates, hence LibScout is not robust against package flattening.

LibPecker [ZDZ+18] improves on LibScout and ORLIS by using internal class dependencies and method invocation relations. LibPecker was concluded by Zhan et al. [ZFL+20] to be the most robust against obfuscation. However, due to it using the package structures for finding possible libraries, it is not robust against dead code removal and package flattening [ZLF+22].

3.3 Binary Code Similarity

The survey done by Haq and Caballero [HC21] gives an overview about past research on binary code similarity. It lists practical applications for binary code similarity, as well as several different techniques and approaches. Furthermore, it shows how binary code similarity evolved from simple byte-level diffing in 1991 proposed by Reichenberger [Rei91] towards the neural network based approaches of recent years, e.g. αDiff [LHZ+18], BinDeep [TJM+21] and the work of Yu et al. [YCT+20].
αDiff calculates a feature vector for each function based on their raw bytes, their relationships with other functions and imported functions. A function's raw bytes are transformed into an embedding vector by a Convolutional Neural Network (CNN). The inter function semantics are represented by the in- and out-degree of the function CG. Due to the fact that an CNN is used for creating the embedding it is not possible to ascertain the exact features that have been learned [HC21, KKC+23].
Yu et al. [YCT+20] use Natural Language Processing (NLP) models to extract semantics from the binary code. It treats basic blocks as sentences and the tokens in them as words in order to apply BERT [DCLT18]. BERT is an influential NLP model structure based on a transformer architecture. The embeddings learned from BERT are then processed in a Message Passing Neural Network (MPNN) [GSR+17] to calculate the graph semantic and structural embedding. These two are combined in a final step with a graph order embedding created by applying a CNN to the adjacency matrix of the CFG.
Kim et al. [KKC+23] investigated features used by various binary code similarity techniques. Based on their results they implemented a similarity measure using syntactic and structural features, e.g. number of specific types of instructions or average basic block size. They use a greedy based selection algorithm for creating an optimal feature set from their list of possible features. Their main goal with that was to create an interpretable binary code similarity measure, showing that uninterpretable neural networks and feature embeddings are not required to get a good performing binary code similarity technique. Their tool *TikNib* outperformed other more complex state-of-the-art approaches. For

evaluation, they created and published their own dataset called *BinKit* consisting of 243,128 binaries, as thy noticed that the lack of such a dataset made it difficult to compare various techniques with each other. Using this dataset and their *TikNib* tool they investigated the effects of different compilers, compiler options, obfuscation and other others on the similarity of the produced binaries.

CHAPTER 4

Breaking Current Framework Detection Techniques

In this chapter we show that the current approaches listed in section 3.1 are not robust against simple obfuscation techniques. For this we took a random app for each framework from our dataset, obfuscated that application and confirmed that the application still can be installed and run, whilst at the same time not being detected any more by the framework detectors. In such a case we call a detection technique broken. We only tested the following framework detection techniques as the other techniques were either closed source or outdated, such that they could not be run reliably or targeted deprecated frameworks and framework versions: Mob Framework Radar [Mob], APK Framework Detector [APKa], Biørn-Hansen et al. [BHGM+22], APK Platform Detector [APKb], Power APK [Pow] and DroidLysis [Dro].
All current techniques, which are shown in table 3.1 function by similar principles mentioned in section 3.1, hence, the same obfuscation approach defeating one detector also defeated others detectors.
The following of this chapter is structured as followed. The sections 4.1-4.4 will each introduce one obfuscation technique and show a proof of concept script using it. All shown techniques have been used to defeat one or more of the above listed framework detectors. Furthermore, in section 4.5 we will summarize our findings and argue what these results mean to the effectiveness and robustness of the framework detection techniques not tested.

4.1 Library Renaming

Most of the current framework detection approaches search for the presence of framework specific libraries in order to detect some of the frameworks. To check the presence of a library they simply search the APK for a file with the corresponding name, i.e.

```
apktool d -f com.zulipmobile_216.apk
cd com.zulipmobile_216

# Rename reference to the libraries
find . -type f -exec sed -i "s/libreactnativejni.so/libobfuscatedjnir.so/g"
↪  {} \;

# Modify System.loadLibrary() calls
sed -i 's/reactnativejni/obfuscatedjnir/'
↪  smali/com/facebook/react/bridge/ReactBridge.smali

# Rename library files itself
find ./lib -type f -name "libreactnativejni.so" -exec sh -c 'mv "$0"
↪  "$(dirname "$0")/libobfuscatedjnir.so" ' {} \;

...
```

Listing 4.1: Bash script applying Library Renaming to obfuscate a React Native APK

libflutter.so, *libreactnativejni.so*, etc. Hence, it is straightforward to defeat this technique. To change the name of the library file three things have to be done. First the library file itself has to be renamed, then all mentions of this library filename in the app have to be changed to the new library filename, afterwards the library name given to the `System.loadLibrary()` call in the application code has to be changed to the new name of the library. The name used in the `System.loadLibrary()` call is slightly different to the actual filename of the library, e.g. a library with filename `libflutter.so` will be loaded by calling `System.loadLibrary("flutter")`, hence, dropping the prefix *lib* and the *.so* file extension.
Listing 4.1 shows an example bash script obfuscating a React Native application. The script unpacks the APK file using *apktool*[1], before using *find*[2] and *sed*[3] commands to find mentions of *libreactnativejni.so* and *reacnativejni* and replace them with *libobfuscatedjnir.so* and *obfuscatedjnir*. We omit the commands used for repacking, realigning and signing of the APK as these are not relevant for obfuscation itself.
The *LibEncryption* obfuscation of *Obufscapk*[4] should also prevent framework detection based on the library file names as it encrypts the library files and moves them with a changed name to the *assets/* directory. However, we found it to not be reliable as it only does this for libraries loaded within static constructors, which is not always the case.

[1] https://apktool.org/
[2] https://linux.die.net/man/1/find
[3] https://linux.die.net/man/1/sed
[4] https://github.com/ClaudiuGeorgiu/Obfuscapk

```
...

# Rename assets/www directory to obfuscate usage of Flutter
mv ./assets/www ./assets/obfuscated
find ./smali -type f -name "*.smali" -exec sed -i 's#www/#obfuscated/#g' {}
↪  \;

# Rename cordova.js to obfuscate the usage of Apache Cordova
mv ./assets/public/cordova.js ./assets/public/obfuscated.js
find . -type f -exec sed -i "s#/cordova.js#/obfuscated.js#g" {} \;

# Rename index.android.bundle to obfuscate usage of React Native
mv assets/index.android.bundle assets/index.obfuscated.bdl
sed -i 's/index.android.bundle/index.obfuscated.bdl/'
↪  smali/com/facebook/react/ReactNativeHost.smali

...
```

Listing 4.2: Bash commands applying Asset Renaming obfuscation against a Flutter application

4.2 Asset Renaming

Another often used detection technique is to look for certain asset files, which can also easily be obfuscated by just renaming these or the directories containing them. Here we first rename the framework specific directories or files before changing all mentions of them in the applications' code.
Listing 4.2 shows some commands, that were successfully used for obfuscating against the tested detection techniques. The script again omits the commands used for unpacking and repacking the APK. Changing asset names is a common technique also implemented by the *AssetEncryption* obfuscation of *Obfuscapk*, however we found this particular implementation not reliably modifying relevant assets.

4.3 Class Renaming

Class names are commonly used for inferring information about an application, however removing these prior to releasing an app is easy as the default obfuscator/compiler R8, that is part of the Android build chain is capable of renaming classes to random strings [Andl]. The investigated framework detection techniques often searched the *AndroidManifest.xml* and *classes.dex* files for mentions of framework specific class names. Renaming classes in Android also requires moving their corresponding class files, as the fully qualified class name corresponds to the directory and filename of the class. Some detectors, like DroidLysis and APK Platform Detector searched the classes and directory hierarchy for parts of qualified class names, e.g. *"com.xamarin"* and *"com/appcelerator/aps"*, obfuscating against such techniques requires moving the entire directory and renaming

```
...

# Rename reference to class in smali code
sed -i
↪  's#com/tns/NativeScriptApplication#com/tns/ObfuscatedScriptApplica#g'
↪  ./smali/com/tns/NativeScriptApplication.smali

# Rename class file
mv ./smali/com/tns/NativeScriptApplication.smali
↪  ./smali/com/tns/ObfuscatedScriptApplica.smali

# Rename all references to the class
find . -type f -not -path "./smali" -exec sed -i
↪  's/com.tns.NativeScriptApplication/com.tns.ObfuscatedScriptApplica/g'
↪  {} \;

...
```

Listing 4.3: Bash commands applying Class Renaming obfuscation against NativeScript applications

all classes inside of them.
We used *Obfuscapk's ClassRename* obfuscator and noticed that the resulting APK crashed upon start, this is due to *Obfuscapk* not properly handling usage of the renamed classes if accessed in framework specific JavaScript code or libraries. Therefore, we implemented our own proof of concept class renaming scripts. The example found in listing 4.3 handles this for NativeScript. First all mentions of the class name *com.tns.NativeScriptApplication* inside the smali code are renamed, before the files and directories containing the class are renamed. Lastly all other references to the class are renamed including usages in *./assets/app/vendor.js*, which *Obfuscapk* did not. Even tough a simple fix this shows that obfuscators need some framework specific tweaks to function properly when applied to non-native Android applications.

4.4 Resource Renaming

The framework detector of Biørn-Hansen et al. [BHGM+22] searches the *AndroidManifest.xml* file for the metadata entry name *"qt_libs_resource_id"*. Simply renaming the name of the metadata entry in all places circumvents detection. The command used for this can be found in listing 4.4. We also tested the implementation of the *ResStringEncrpytion* obfuscator of the *Obfuscapk*, but it failed to rename the relevant resource names.

```
# Rename qt_libs_resource_id id
find . -type f -exec sed -i 's/qt_libs_resource_id/obfuscated/g' {} \;
```

Listing 4.4: Bash command applying resource renaming against Qt Framework applications

4.5 Summary

To summarize we have defeated all the following framework detectors, representing the state-of-the-art approaches: Mob Framework Radar [Mob], APK Framework Detector [APKa], Biørn-Hansen et al. [BHGM+22], APK Platform Detector [APKb] Power APK [Pow] and DroidLysis [Dro].
The obfuscation we introduced prevented them from detecting the following frameworks:

- Apache Cordova
- Capacitor
- Flutter
- Ionic
- NativeScript†
- Qt Framework
- React Native
- Titanium SDK
- Xamarin*

We also tried detecting applications using Expo, but no detector found any Expo related features in the applications. However, as Expo is based on React Native, corresponding features were found. Applying the same obfuscations as against React Native Applications was successful in obfuscating the React Native features in the Expo applications.
We could not prevent Power APK from detecting the Xamarin framework as it searched for fully qualified class names starting with *"com.xamarin"* and renaming these classes resulted in the application failing to launch. This is due to a native library, that is part of the Xamarin framework, accessing Xamarin classes using their fully qualified class names in a way that can not simply be changed with a string replacement operation on the native library. Using reverse engineering we also failed to precisely identify how the calls to these classes are made in the native library. However, we were able to find relevant code snippets in Xamarin's source code. Listing 4.5 shows the JNI name of one of the classes in the *com.xamarin* class path, which we failed to rename, in a way to make the app still executable.
In order to detect the NativeScript framework DroidLysis searches the directory hierarchy for *"com/tns"*. However, moving the entire directory and renaming all the classes within it correspondingly, results in the app crashing upon start up as now NativeScript specific JavaScript code cannot find the main application class via the JNI any more.
This again shows that class renaming, even though a simple obfuscation technique can not always be directly applied when an application is build with a framework. It is therefore unlikely that class names in applications using Xamarin and NativeScript are obfuscated,

†DroidLysis technique not broken
*Power APK technique not broken

```
1  namespace Java.Interop {
2
3    [JniTypeSignature (JniTypeName)]
4    /* static */ sealed class ManagedPeer : JavaObject {
5
6      internal const string JniTypeName =
   ↪  "com/xamarin/java_interop/ManagedPeer";
7
8      static  readonly    JniPeerMembers  _members        = new
   ↪  JniPeerMembers (JniTypeName, typeof (ManagedPeer));
9
10     static ManagedPeer ()
11     {
12       ...
```

Listing 4.5: *ManagedPeer* class in the Xamarin source code for the native library. Line 6 shows the JNI class name string.

making this a reliable artefact for detecting NativeScript and Xamarin applications. All other listed frameworks were successfully obfuscated without access to their source code.

The obfuscation techniques applied by us are very basic string manipulation and file moving, applied to APK files that were unpacked and packed using *apktool*. Obfuscators applied at compile time have more options and can perform complex obfuscations in a more robust manner. Working our approach into a robust production ready obfuscation tool would require more polishing, nevertheless it shows that it is trivial to circumvent current framework detection techniques even without deep understanding of application code using simple string manipulations.
Although we have only tested a subset of the framework detection techniques, the findings and conclusions of this chapter can be applied to all techniques as they work by similar principles.

Whilst it is easy to obfuscate the used frameworks, we noticed that existing obfuscators might fail to produce runnable results, due to them being designed for native Java/Kotlin applications, therefore, not performing necessary framework specific obfuscation steps, e.g. *Class Renaming* not renaming usages inside framework specific JavaScript code as mentioned in section 4.3. As a result the current framework detection techniques might still deliver good results as artifacts used for detection are not getting obfuscated due to lacking support of the obfuscators.

CHAPTER 5

Framework Detector

This chapter introduces the framework detector developed as part of this thesis and explains how it functions and which considerations have been made during its development. The structure of this chapter is as follows: Section 5.1 lists the frameworks supported by our detector. The next section 5.2 explains how the dataset, used for training and evaluation, has been build and how many applications for each framework it contains. The framework detector itself will be explained in section 5.3, we explain the tools it uses, as well as the modifications we made to them.

5.1 Supported Frameworks

Over the years there have been many different frameworks available for Android. The rapid development of the mobile market led to many different approaches. Most did not become popular enough to persist. We decide to include the following 16 frameworks:

- Apache Cordova [Apaa]
- Apache Flex [Apab]
- Capacitor [Cap]
- Expo [Exp]
- Flutter [Flu]
- Framework7 [Fra]
- Ionic [Iona]
- Kivy [Kiv]
- NativeScript [Nat]
- Qt Framework [QtF]
- React Native [Reab]
- Solar2D [Sol]
- Titanium SDK [Tit]
- Unity [Uni]
- Uno Platform [Uno]
- Xamarin [Xam]

We decided on these frameworks as they are still in active development or have been relevant in the near past, with many apps still using them. We excluded frameworks for which we could not find any available apps using them. This affected Meteor [Met] and Rhodes [Rho]. In addition, we excluded any frameworks used for only building Web apps as these do not produce an APK, but are run using a browser. Furthermore, we focused on CPDF frameworks but nevertheless included some popular game engines, like Unity, as these are specialized frameworks. Based on public surveys [Staa] and frameworks investigated by other research we conclude that the list of frameworks chosen contains the currently relevant frameworks [BHGM+22, BHGG+19, AM16].
As many frameworks extend each other one difficulty is to decide when something can be considered its own framework and when just a version of another. For example Expo is similar to React Native supporting and relying on many React Native modules. Similar Apache Cordova is the basis for many other frameworks such as Capacitor, Ionic and Framework7. With Ionic itself again being based on Capacitor [MS21, Ionb, BHGG18]. This question can never be answered with absolute certainty. The thesis adopts an approach where a framework is treated separately, even if it extends another. This separation is based on the extent to which a framework brings significant differences and improvements compared its parent. Additionally, the development process using the framework may differ enough to justify this separation. Especially as a framework matures and gets developed further it differentiates more from the framework it extends. To avoid confusion when comparing this list with other research it is to be emphasised that renaming and rebranding of frameworks is rather common. For example Apache Cordova is developed from an open source fork of Phone Gap, before Phone Gap was deprecated in favour of the development of Apache Cordova [Phoc, Phoa, Phob].

5.2 Dataset

For the creation of the dataset we used mainly two approaches. Our first approach was to manually search various frameworks' websites for a list of sample or showcase applications´. We scraped the applications' information both manually and automatically in order to generate the basis of our dataset. Using this technique we were able to retrieve 527 possible apps, for 335 of which we were able to retrieve a link to the Play Store to download the apps from. We were able to download 142 apps directly from the Play Store. As not all apps could be downloaded directly from there, 127 of these were downloaded from the Androzoo database [ABKLT16], which also have been downloaded from the Play Store. Due to constant changes in how apps are distributed via the Play Store, the way they can be downloaded is not always reliable. The other 66 applications we were not able to download.
We could not find 192 applications on the Play Store, but were able to obtain their source code. We had to build them manually, setting up a development environment for each framework and compiling them. However, we were not able to build most apps and samples, as only 12 of them could be built. The most common reasons for build failures were dependencies on old non-existing libraries and modules, hard coded environment

variables, missing Keystore files required for signing the APKs and missing configuration files, e.g. for Google services. Some problems could be solved by looking at each app and sample individually and manually resolving the errors occurring during the build process. However, we considered this out of scope.

As an alternative approach for finding applications using particular frameworks, we searched the F-Droid data repository [1]. In order to determine the framework, only the apps' most recent build metadata is used. For each of the 4,363 apps in the repository there exists a metadata file containing basic information about the app as well as steps for building it. These metadata files were parsed and search for string signatures indicating the framework used by the app. Each string was only searched for in the specific metadata entry type, i.e. *sudo*, *scanignore*, etc. as specified in table 5.1. An overview over the different entry types of the metadata file can be found in the F-Droid documentation[2]. The signatures used can be seen in table 5.1. One matching signature suffices to identify the framework, except to for Apache Cordova, Capacitor and Ionic, as these extend each other. If an Ionic signature is found it takes precedence over any Apache Cordova or Capacitor signature. Similarly, Capacitor signatures overrules Apache Cordova signatures. This is necessary as apps using the Ionic and Capacitor framework might use the same modules, parts and commands as the framework they are based on.
Especially frameworks using NPM[3] often contained relevant signatures that could be attributed to a particular framework. A downside of this approach is that many applications only made use of a *gradle* build file, hence, their metadata files contained no relevant information for attributing the framework. As these build files and the source code of the applications were not parsed, no framework information could be retrieved for these apps. Future work might want to look at the source code and files relevant to the build process of each application in order to determine the framework for more applications. Table 5.2 shows the number of apps from F-Droid, of which we were able to determine their framework.

An overview over the final dataset can be found in table 5.3, showing the number of apps we could retrieve an APK file for, as well as the source we got the APK from.
We could not download APKs for all the apps found in the F-Droid data repository as some apps have been deprecated and taken down with their source code and old build information still remaining in the repository.

[1] https://gitlab.com/fdroid/fdroiddata
[2] https://f-droid.org/docs/Build_Metadata_Reference/
[3] https://www.npmjs.com/

Framework	Signatures
Apache Cordova	• *prebuild* entry starting with *"cordova platform add android"* • Installation of *"cordova"* using *npm* in *sudo* • *build* entry starting with - *"cordova compile android"* - *"cordova build android"*
Capacitor	• *prebuild* entry starting with - *"npx cap sync android"* - *"npx cap"*
Flutter	• *build* entry starting with - *".flutter/bin/flutter build apk"* - *"./flutterw build apk"* - *"$$flutter$$/bin/flutter build apk"* - *"$$flutter$$/bin/flutter -v build apk"* - *"flutter build apk"* • *scandelete* entry starting with *".flutter"* • *scanignore* entry starting with *".flutter/bin/cache"* • *prebuild* entry starting with *".flutter/bin/flutter"* • *srclibs* entry starting with *"flutter@"*
Framework7	• Installation of *"framework7-cli* using *npm* in *sudo*
Ionic	• *prebuild* entry starting with *"ionic cap"* • *prebuild* entry equals *"yarn run ionic capacitor sync android"* • Installation of *"@ionic/cli"* using *npm* in *sudo* • Instalaltion of *"ionic"* using *npm* in *sudo* • *build* entry starting with *"ionic cordova build"*
Kivy	• Installation of *"kivy"* using *pip* in *prebuild* • *build* entry containing: - *"buildozer android release"* - *"./buildozer.run release* • Installation of *"buildozer"* using *pip* in *build*
Meteor	• Installation of *"meteor"* using *npm* in *sudo*
NativeScript	• *srclibs* entry starting with *"NativeScript_"* • Installation of *"nativescript"* using *npm* in *sudo* • *build* entry starts with *"pushd $$NativeScript_NativeScript$$"*

Framework	Signatures
Qt Framework	• *build* entry starting with - *"export QT_BUILD"* - *"export QT_VERSION"* - *"$Qt5_android"* - *"git clone https://code.qt.io/qt/"* - *"$QT5_arm/androiddeployqt"* • *build* entry containing: - *"QT5_DIR"* - *"https://download.qt.io/"* • *rm* entry equals *"qt"*
React Native	• *scanignore* entry containing *"node_modules/react-native/android"* • Installation of *"react-native-cli"* using *npm* in *sudo*
Titanium SDK	• *build* entry starting with *"ti build"*

Table 5.1: Signatures used for determining the framework based on the apps' F-Droid metadata file. One of the signatures matching suffices to identify a framework.

Framework	Total
Apache Cordova	6
Capacitor	4
Flutter	184
Ionic	7
Kivy	7
NativeScript	1
Qt Framework	9
ReactNative	25
Total	243

Table 5.2: Numbers of F-Droid apps with known framework

5.3 Detection

In the following section 5.3.1 we explain the structure of the detector and how it works. Section 5.3.2 explains the process of finding relevant functions, used for identifying each framework, before we discuss the limitations of our detector in section 5.3.3.

Framework	Manually Built	Downloaded from Play Store	Downloaded from F-Droid	Total
Apache Cordova	0	0	6	6
Apache Flex	0	3	0	3
Capacitor	0	0	4	4
Expo	3	17	0	20
Flutter	0	0	184	184
Framework7	0	75	0	75
Ionic	0	5	7	12
Kivy	0	0	7	7
NativeScript	0	44	1	45
Qt Framework	0	0	9	9
React Native	8	69	25	102
Solar2D	0	34	0	34
Titanium SDK	1	0	0	1
Unity	0	6	0	6
Uno Platform	0	1	0	1
Xamarin	0	15	0	15
Total	12	269	243	524

Table 5.3: Number of APKs grouped by framework and their source

5.3.1 Architecture

Our framework detector detects functions based on features extracted from the binary OAT representation using the *TikNib*[4] tool developed by Kim et al. [KKC+23]. The applications' OAT representation is created using the approach described by Bleier and Lindorfer [BL23]. We decided to use their approach as it allows generating the native binary representation for Android applications from their APK file using Androids *dex2oat* compiler. The additional compilation step optimizes the code, thus normalizing it, weakening the effect of some obfuscation techniques. The decision to use *TikNib* is based on its good results on binary similarity across different compilers, compiler options and architectures outperforming recent state-of-the-art techniques, whilst not relying on in-transparent machine learning approaches.

Figure 5.1 gives an overview how an application is analyzed using our detector. First *dex2oat* is used to generate each app's OAT file from its APK. This step just takes a few seconds for all apps. After that the OAT file is taken to *TikNib*, which first runs *IDA Pro*[5], in order to decompile it and determine the functions, the CG, as well as the CFG. In the next step we filter out all functions, whose fully qualified name starts with *BakerReadBarrierThunk* or *androidx::* as we found that these functions did not contribute to distinguishing different frameworks, as they occurred across most applications. By

[4] https://github.com/SoftSec-KAIST/tiknib
[5] https://hex-rays.com/IDA-pro/

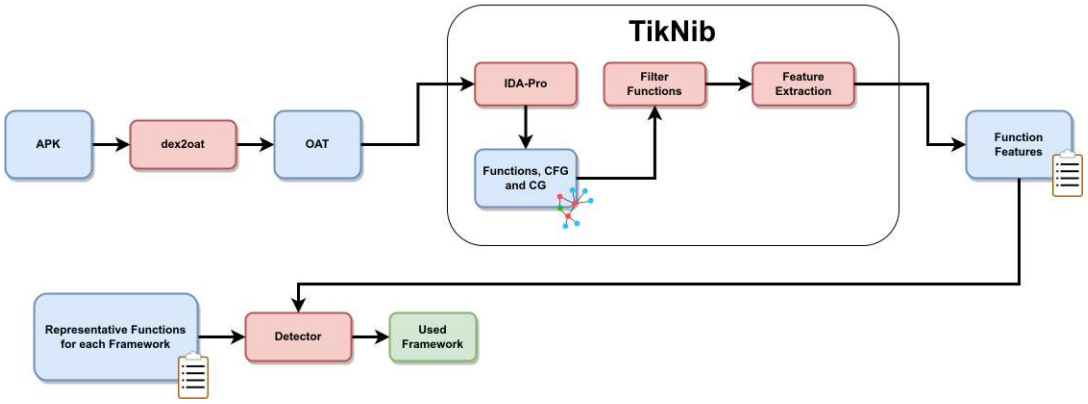

Figure 5.1: Overview of the detection workflow

removing them we saved computation resources in all following steps as the number of functions was reduced. In the last step of *TikNib* the actual features for each function were extracted and written to a *pickle* file.

For detection these pickle files are read in, and all functions are compared to the frameworks' reference functions. The framework with the most matching functions will be selected. How representative functions for each framework are selected is explained in section 5.3.2 later in this chapter.
We experimented with normalizing the number of found functions over the number of each framework's reference functions. However, we found that this decreased the detection rate slightly whilst almost doubling the number of false positives. Furthermore, we looked into having special cases for handling related frameworks, such as the ones based on Apache Cordova. For this, the child framework with the most matching functions would be selected instead, if the number of matching reference functions was higher than a percentage of the parents matching reference functions. The idea behind this approach was that related frameworks will contain functions of the frameworks they are based on, hence if the number of matching functions in a child framework is sufficiently large it could indicate that the child framework instead of the parent framework is used. However, we found this to not have any significant impact, as regardless of the percentage chosen this mechanism lead to roughly the same amount of correct and incorrect decisions.

Scoring Metric

The metric used to determine the similarity of two functions is the average of the relative differences between all features of the two functions. This is the same as proposed by Kim et al. [KKC+23] with the relative difference between a feature f of functions A and B being calculated as follows:

$$\delta(A_f, B_f) = \frac{|A_f - B_f|}{|\max(A_f, B_f)|} \tag{5.1}$$

The similarity between two functions is defined by the average of all relative differences between the functions' features as follows.

$$\text{sim}(A, B) = 1 - \frac{\delta(A_{f_1}, B_{f_1}) + \cdots + \delta(A_{f_N}, B_{f_N})}{N} \tag{5.2}$$

The similarity score lies in the range of $[0, 1]$ with 1 corresponding to a complete match and 0 being the greatest difference. Only features occurring in both functions are used for calculation of the similarity score, i.e. if a feature is only defined for one of the two functions it will not be used for calculating the similarity between them. During detection, we deem two functions matching or similar if their similarity value is greater than 0.95. This scoring metric was also used in the study by Kim et al. [KKC+23], where it showed good results.

Features

For detection, three types of features are used: instruction features, CFG features, and CG features. Examples for instruction features are the total number of arithmetic instructions per function or the average number of shift instructions per basic block in a function. CG features for example are the number of callee and callers for a function and examples for CFG features are the number of loops or the max depth of the CFG.

Modifications to TikNib

We had to make several modifications to *TikNib* as it was developed for evaluating binary code similarity across different compilers, compiler options and architectures. For example, it expected the filenames of the given binaries to contain the used compiler, compiler options as well as the architecture to automatically output the results for the original authors' different research questions. We modified *TikNib* such that it can be given any binary file, to create an output *.pickle* file containing a list with all the found functions and their features within it. In addition, we removed the original criteria for filtering out functions as they were not intended for generating features for all functions. We also disabled the extraction of the data features, as we noticed that these were always 0 when being run on OAT files. As each step in *TikNib* generated an intermediate *pickle* file we removed all unnecessary information in each step in order to keep the size of these pickles small.
Kim et al. also looked into how type information changes the effectiveness of their binary code similarity technique. We removed the extraction of these type features as it relied on debug symbols, which were not available for most functions in the generated OAT files. Furthermore, their approach still showed good results without type information. As type information in general is available in the application's Dalvik code, possible future work could look into extracting and adding type information the extracted functions.
TikNib also contained a feature selection procedure used for learning and selecting the most significant features. However, we did not use it for our detector and instead made use of all features.

5.3.2 Function Selection

In this section we will explain how the functions representing each framework were found and selected. We used a semi-automatic approach, where the first step produces suggestions for relevant functions based on their fully qualified names, which then have been tweaked and refined in a second manual step.

As the used binary code similarity metric gives us pairwise similarities between two functions and most applications have more than 15,000 functions, it is unfeasible to calculate similarity scores for all function pairs, in order to determine similarity clusters for each framework. Therefore, we had to pre-select relevant functions. For this we used the fully qualified function names to find functions that occurred in applications of the same framework, but were not present across frameworks. For this we used all applications in the dataset for which we could extract features using *TikNib*. Relying on the fully qualified names to identify functions is not ideal, because obfuscation of class and function names will alter these. Therefore, we manually checked and removed proposed function names that were obfuscated. As we used a number of open source apps taken from F-Droid, for which the use of obfuscation techniques is rather unlikely we can safely assume that we are able to gather enough relevant function names without being hindered by function and class name obfuscation. Furthermore, as seen in chapter 4 it is likely that most class name obfuscators are not applied to framework classes, as they lack the framework specific handling in order to produce correctly functioning apps.

The selection of relevant function names was done in multiple steps listed in the following.

Step 1: Fetching of the function names for each application.

Step 2: Identification of *inter-framework* functions. These are common functions that are present in multiple frameworks. A function is considered common if it occurs in at least 50% of the applications with the same framework. For each pair of non-related frameworks we checked if their common functions occurred in the other frameworks common functions. We required these functions to occur in at least half of the applications of each framework in order to not identify functions only occurring in a few applications. The two groups of related frameworks considered by us are Expo and React Native, as well as Apache Cordova, Ionic, Capacitor and Framework7.

Step 3: Removal of *inter-framework* functions. The *inter-framework* functions identified in the previous step are now removed together with the function names listed on a manually created black list. This list is based on observations made during development and testing of the detector. It contains concrete function names as well as regular expression matching function names that we noticed to have no or a negative impact on the detection capabilities.

Step 4: Identification of relevant functions. We selected the 20 most common function names for every framework. Each function can only occur at most once per application. Following the 20th most frequent function, up to 10 additional functions would be included if they shared the same occurrence frequency as the 20th function. This introduced

a degree of randomness in choosing functions with equal frequencies after the 20th position, necessitating a manual process for selecting desirable functions and eliminating undesirable ones. The functions manually added, were independently chosen in advance before selecting the 20 most common functions. We found using more than 20 functions at this step to increase the time needed for detection without increasing the detection performance.

Step 5: Feature selection. In this step we chose all the features associated with the function names deemed relevant. In instances where multiple distinct features were associated with the same function name, we included all of them. However, such occurrences were infrequent, suggesting that function names served as an effective tool for our selection purposes.

Step 6: Removal of similar functions. Using the features from the previous step we calculate the pairwise similarity between all functions using the score introduced in section 5.3.1. We excluded functions that exhibited a high degree of similarity to each other (sim > 0.9). This step aimed to eliminate redundancy among relevant functions within each framework and to filter out functions with similar features across different frameworks.

After these steps we have the relevant function features for each framework, which can be used for detection.

5.3.3 Limitations

One of the biggest drawbacks of our detector in comparison to the other framework detection approaches is the long runtime. As each application has to be decompiled using *IDA Pro* before each function's features can be extracted. We ran *IDA Pro* with a timeout of 45 minutes, which was reached by several applications. For most applications the decompilation and feature extraction using *IDA Pro* took around 30 minutes. Furthermore, big applications can not be analyzed using our detector, as we noticed OAT files bigger than 30MB had a very high chance of running into the timeout, as a result we did not try to decompile applications with OAT files larger than 100MB. In the *default* dataset, 49 apps (9.35%) had OAT files larger than 100MB. This size constraint limits the applications our detector can be applied to, one possible mitigation for this could be to skip the decompilation step and instead extract the features for each function from the *oatdump* output directly. The *oatdump* can be generated during the generation of the OAT files and contains the native binary code side-by-side with the Dalvik function it was generated from. These generate much faster, however we leave it to future work to investigate if the necessary features can be extracted from just the *oatdump* file directly.
The *dex2oat* compiler, that is used during the creation of the OAT files, has a *–dump-cfg* option. We looked into the created CFGs but found them to not be usable for our purpose, as they only contain the control flow between the basic blocks of within each function for different stages of the compilation process.

CHAPTER 6

Evaluation

In this chapter, we assess the framework detection technique introduced in chapter 5 as well as the following framework detectors introduced in section 3.1: APK Framework Detector [APKa], Mob Framework Radar [Mob], DroidLysis [Dro] and the detector by Biørn-Hansen et al. [BHGM+22]. We decided to use these detectors for the evaluation as their source code is publicly available, they support recent frameworks and can be run without outdated distributions and software packages. We evaluate each technique on our dataset introduced in section 5.2 as well as with two different obfuscation techniques applied. Namely, the obfuscations developed by us in chapter 4, as well as control flow obfuscation using Obfuscapk. In addition, we also evaluate the detectors with both obfuscation approaches applied.
The chapter is structured as follows. First we will explain the methodology of the evaluation and how the obfuscation was done in section 6.1, before discussing the results in section 6.2. Threats to the validity will be discussed in section 6.3.

6.1 Methodology

To facilitate the evaluation process, we configured the current framework detection techniques as Docker images. This enables their seamless and reproducible utilization, while remaining system-independent. Our detector is not run in a Docker container, and runs directly on the host system, but it can be dockerized if needed. We decided against implementing it as a Docker container, as running directly on the host system made the development of the proof-of-concept easier. The evaluation itself is done using a Python program that takes each APK file and runs each detector on it. Our detector takes an APK and creates its OAT binary representation, from which it extracts the features for each function using *TikNib*. The function features for each application are stored in a pickle file for later use. As extracting the features is a time-consuming process and

only needs to be done once per APK, we have separated the feature extraction from the evaluation.

We collected the framework identified by each technique in the applications and tallied the results. Detection of the correct framework is counted as a true-positive, and detection of the wrong framework as false-positive. Each detector except for the APK Framework Detector and our detector can return more than one found framework per application. Hence, if more than one framework was detected by a technique each wrong result counts as a false-positive, and the correct one as a true-positive. The maximum number of applications, where multiple frameworks have been detected was 25 with the Mob Framework Radar on the *default* dataset. On the *obfuscated-framework* and *obfuscated-both* dataset this occurred only a total of three times. Cases where no framework could be detected, or the technique failed for some reason, i.e. timeouts or exceptions during execution, were not counted as a false-positive. However, the number of applications where no framework was detected or the detector failed are also noted in the results. We did not count results or failed detection attempts as a false-positive, as no result is different from returning a wrong result. Only applications using a framework supported by the used detector, were evaluated, i.e. if a detector does not support detecting framework A no applications using framework A will be given to the detector.

We evaluate 4 different datasets that are all based on the *default* dataset introduced in 5.2. How these datasets were created from the *default* dataset will be explained in the following.

Obfuscate-Framework Dataset

The *obfuscate-framework* dataset was created by applying the concrete obfuscation techniques developed and shown in chapter 4 to the *default* dataset. This dataset represents the application of class and function name obfuscation, which is common for Android applications, as well as the renaming of library and asset files. However, we did not use an existing obfuscator for this as we noticed in chapter 4 that naively renaming classes and functions without taking special care for classes used by frameworks will result in non-functioning apps. We therefore decided to use the obfuscation techniques and scripts shown previously, as they were engineered to handle framework specifics whilst renaming classes and functions used by the frameworks.
Of the 524 APKs in the *default* dataset, 388 obfuscated APKs could be produced using this approach. Only 10 apps failed to produce an APK file, the other apps were not obfuscated as they used frameworks (Framework7, Kivy, Solar2D, etc.), for which we did not have an obfuscation script. It is to be noted that it has not been validated if the created APKs could still be installed and executed. This was only tested for the applications used for developing the obfuscation scripts in chapter 4.

Detector	True-Positive	False-Positive	# apps (Frameworks)
Our Detector	288 (54.96%)	33 (6.3%)	524 (16)
Our Detector Apps without features ignored	288 (89.72%)	33 (10.28%)	321 (15)
APK Framework Detector [APKa]	284 (92.51%)	23 (7.49%)	307 (4)
Mob Framework Radar [Mob]	292 (87.95%)	33 (9.94%)	332 (7)
Biørn-Hansen et al. [BHGM+22]	216 (57.14%)	15 (3.97%)	378 (9)
DroidLysis [Dro]	271 (78.78%)	4 (1.16%)	344 (6)

Table 6.1: Number of True-Positives, False-Positives, total apps and number of detectable frameworks for the different detectors on the default dataset. Apps without features ran into a timeout or an error during the feature extraction with *TikNib*. We found DroidLysis to work unreliable, giving different results for each run.
Green cells indicate the best performance per column.

Obfuscate-CFG Dataset

As our detector relies on features based on the CFG and CG, as well as assembler instructions within each function we decided for the second dataset *obfuscate-cfg* to use Obfuscapk with the following obfuscation options: ArithmeticBranch, CallIndirection, Reflection, Reorder, Goto and Nop. These are all obfuscation options that change the CFG and CG of the application and insert additional instructions into the functions, possibly changing the function features. Therefore, we suspect this dataset to perform worse on our detector than the *default* or the *obfuscate-framework* dataset.
478 APKs could be obfuscated using this approach. 23 Apps of the 46 that failed to be obfuscated were using React Native. We did choose Obfuscapk as it seems like a mature obfuscator, that in contrast to other obfuscation tools can directly be applied to APK files, making it suitable for our use case. However, the Obfuscapk project has been archived on GitHub on November 15 2023 and as the time of writing it is unclear if a fork of this project will continue.

Obfuscate-Both Dataset

This dataset combines the obfuscation approaches of both the *obfuscate-framework* and *obfuscate-cfg* dataset and contains 356 APK files. We decided to add this dataset to the evaluation as it combines techniques that in theory should harm existing approaches as well as our approach to framework detection.

6.2 Results

The results for the different detectors evaluated on the default dataset can be found in table 6.1. The different number of total applications for each detector stems from the fact that they support different frameworks, and applications using frameworks not

supported by the detectors were ignored.
The APK Framework Detector performs the best with a true-positive percentage of 92.51%, this means it correctly detects the framework for most of the applications. On the other hand it does only support detecting 4 different frameworks, which is the least out of all the detectors evaluated. With a true-positive percentage of 54.96% our detector performed the worst. This is the case due to the fact that the features of 203 (38.74%) applications could not be extracted within the 45 minute timeout set by us or ran into an error during the extraction. However, when only looking at the performance for applications, where features could successfully be extracted, we see that with a true-positive percentage of 89.72% it performs well. This shows the function features are suitable for detecting the framework used in an application. However, future focus should be set on improving the process of extracting these features faster and more reliable. The false-positive percentage of our detector is not too bad, but with 10.28% still the worst of all the tested detectors. This is to be expected as our detector does not rely on finding static artifacts, like specific directory, file or class names, but instead relies on features extracted from the applications' code. The majority of these false-positives are due to applications using Ionic and Capacitor being detected as the related Framework7 and Framework7 applications as Apache Cordova, as well as 4 of the Xamarin applications being falsely identified as using Flutter. Ionic, Capacitor, Apache Cordova and Framework7 are related frameworks.
The false-positive percentage of the Mob Framework Radar can be explained by it being able to return multiple detected frameworks, whilst supporting related frameworks like React Native and Expo, as well as Apache Cordova and Capacitor. As related frameworks may contain the same files, this can lead to them both being detected, therefore increasing the false-positive percentage. The same applies to the detector of Biørn-Hansen et al.
The fact that DroidLysis performs similar to the other detectors surprises, as it was not purpose build for detecting frameworks and only relies on a single indicator for detecting each framework. Nevertheless, we found it to work unreliable, giving different results for each run. This issue arises from the execution of other time-consuming and complex analyzes, leading to crashes, errors and timeouts. Modification of DroidLysis, removing these, only keeping the analysis steps used for the detection of frameworks could lead to a better reliability.

6.2.1 Differences based on Source

Table 6.2 contains the number of true-positives, false-positives, as well as the total number of applications for each source of apps. The different sources of applications were discussed in section 5.2. We count apps downloaded from the Androzoo dataset to the Play Store source. We can see that our detector performed the best on apps taken from the F-Droid store and worst for apps from the Play Store. This is due to 162 applications from the Play Store failed to produce any features within the set timeout period. This hints towards apps on the Play Store being more complex and therefore more resource intensive to decompile and analyze. Possible obfuscation of the applications from the Play Store could also play a role as they could make decompilation and analysis harder

Source	True-Positive	False-Positive	# apps
PlayStore	86 (31.97%)	21 (7.81%)	269
F-Droid	197 (81.07%)	12 (4.94%)	243
Self-Build	5 (41.67%)	0	12

Table 6.2: Number of True-Positives, False-Positives and number of total apps per source for our detector on the default dataset. The poor performance for apps from the Play Store is due to 162 apps failed to produce any features within the set timeout period.

Source	True-Positive	False-Positive	# apps
PlayStore	86 (80.37%)	21 (19.63%)	107
F-Droid	197 (94.26%)	12 (5.74%)	209
Self-Build	5 (100.0%)	0	5

Table 6.3: Number of True-Positives, False-Positives and number of total apps per source for our detector on the default dataset with apps where no features could be extracted within the set timeout period being ignored.

and more time-consuming.
When ignoring all apps without any features available as shown in table 6.3 we can still see that the performance of Play Store apps is worse than the apps from the other two sources, be it to a smaller degree. Except for the detector of Biørn-Hansen et al. this could also be observed with the other detectors. For the Biørn-Hansen detector the performance on the apps taken from F-Droid suffers due to the poor detection of Flutter applications (TP: 25.0%), which make up the majority (75.72%) of applications in the F-Droid dataset. Nevertheless, with a true-positive percentage of 88.72% and a false-positive percentage of 3.76%, it performs well on Play Store applications.

6.2.2 Differences Based on Framework

The detection performance of our detector across different frameworks can be found in table 6.4. The table contains the number of true-positives, false-positives and the number of total applications of our detector for the default dataset, broken down by the applications' frameworks. The percentages for the true-positives are calculated using the number of true-positives and total applications for each framework. The same applies to the false-positives, with the result that these percentages will not add up to 100%. The poor performance for frameworks like Expo, NativeScript, Framework7 and Solar2D can again be explained by the fact that for many applications the feature extraction failed due to a timeout during decompilation. Table 6.5 shows the detection rates per framework when ignoring applications, without features. Here we can see that the detection performance varies significantly between the different frameworks. We looked into what apps contributed to the false-positives for each framework and found most of them to stem from related frameworks. For example, all the 7 Ionic applications were detected as using Framework7, the 6 false-positives of Apache Cordova were actually

Framework	True-Positive	False-Positive	# apps
Apache Cordova	3 (50.0%)	6 (100.0%)	6
Apache Flex	1 (33.33%)	2 (66.67%)	3
Capacitor	0 (0%)	0 (0%)	4
Expo	0 (0%)	0 (0%)	20
Flutter	171 (92.93%)	8 (4.35%)	184
Framework7	46 (61.33%)	13 (17.33%)	75
Ionic	0 (0%)	0 (0%)	12
Kivy	7 (100.0%)	0 (0%)	7
NativeScript	8 (17.78%)	1 (2.22%)	45
Qt Framework	8 (88.89%)	0 (0%)	9
ReactNative	24 (23.53%)	1 (0.98%)	102
Solar2D	19 (55.88%)	0 (0%)	34
Titanium SDK	0 (0%)	0 (0%)	1
Unity	1 (16.67%)	2 (33.33%)	6
Uno Platform	0 (0%)	0 (0%)	1
Xamarin	0 (0%)	0 (0%)	15
Total	288 (54.96%)	33 (6.3%)	524

Table 6.4: Number of True-Positives, False-Positives and total number of apps broken down per framework for our detector on the default dataset.

using Framework7 and Capacitor. All of these frameworks are related. Furthermore, the application using Expo was detected as using React Native, which is also a related framework. This hints towards, our selected function features not being perfectly suited for differentiating between related frameworks as they share to many similar functions. However, to confirm this more applications for each framework are required, as it may be possible to find framework exclusive functions with a large enough dataset. Detecting related frameworks as a framework family without differentiating between them can, depending on the detector's use case, be a viable strategy.

6.2.3 Obfuscation Resilience

We will now look at the different detectors' resilience against obfuscation. Table 6.6 shows the number of true-positives and false-positives for the different detectors on the different datasets. In addition, figure 6.1 shows the number of true-positives and maximum number of detectable apps per framework on the given dataset for the different detectors as a bar chart. The obfuscations applied to each dataset are listed in section 6.1.
The results for the *obfuscated-framework* dataset show the effects of the library, asset, class, and resource renaming obfuscations introduced in chapter 4. We see that they prevent all detectors except of our detector to detect the framework of most apps. That these obfuscations have no big impact to our detector was to be expected as they do not modify the applications' code with exception to renaming functions and classes. However, the extent of their effectiveness against the other detectors is surprising, given their basic nature and possible incomplete implementation, i.e. they do not obfuscate all features

Framework	True-Positive	False-Positive	# apps
Apache Cordova	3 (75.0%)	6 (150.0%)	4
Apache Flex	1 (100.0%)	2 (200.0%)	1
Capacitor	0 (0%)	0 (0%)	3
Expo	0 (0%)	0 (0%)	1
Flutter	171 (98.84%)	8 (4.62%)	173
Framework7	46 (80.7%)	13 (22.81%)	57
Ionic	0 (0%)	0 (0%)	7
Kivy	7 (100.0%)	0 (0%)	7
NativeScript	8 (100.0%)	1 (12.5%)	8
Qt Framework	8 (100.0%)	0 (0%)	8
ReactNative	24 (92.31%)	1 (3.85%)	26
Solar2D	19 (100.0%)	0 (0%)	19
Titanium SDK	-	-	0
Unity	1 (100.0%)	2 (200.0%)	1
Uno Platform	0 (0%)	0 (0%)	1
Xamarin	0 (0%)	0 (0%)	5
Total	288 (89.72%)	33 (10.28%)	321

Table 6.5: Number of True-Positives, False-Positives and total number of apps broken down per framework for our detector on the default dataset with apps where no features could be extracted within the set timeout period being ignored.

used by the different detectors.
The high number of false-positives of the APK Framework Detector is due to it classifying all applications it can not identify the framework for as a native Android application. If the detector were to instead indicate that it is uncertain about the framework used in the app, the number of false-positives would be 0.

When now looking at the results for the *Obfuscated-CFG* dataset, we see that the true-positive percentage of our detector reduces to 30.33%, whilst the percentage of false-positive increases to 8.79%. The majority of the false-positives stems from incorrectly assuming an application using React Native, interestingly at the same time no React Native application could be successfully detected any more. When only looking at applications with features available, Flutter frameworks can still be detected correctly 97.74% and Solar2D 92.86% of the time, whilst detection of the other frameworks almost completely fails. The changes to the performance of the other detectors can mostly be attributed to changes in the number of total apps and apps per frameworks, due to the obfuscation failing for some applications.

When applying both obfuscations, represented by the *obfuscated-both* dataset, we can see their combined effects. In this scenario, our detector stands out as the only one capable of consistently identifying the framework used in an application a significant number of times.

Detector \ Dataset	Default		Obfuscated Framework		Obfuscated CFG		Obfuscated Both	
	TP	FP	TP	FP	TP	FP	TP	FP
Our Detector	54.96%	6.30%	55.67%	6.44%	30.33%	8.79%	37.08%	6.18%
Our Detector Apps without features ignored	89.72%	10.28%	89.63%	10.37%	60.92%	17.65%	77.19%	12.87%
APK Framework Detector	95.51%	7.49%	2.32%	97.68%	95.04%	4.96%	2.15%	96.42%
Mob Framework Radar	87.95%	9.94%	4.0%	0.92%	90.10%	7.59%	3.69%	0.67%
Biørn-Hansen et al.	57.14%	3.97%	2.17%	0.54%	55.52%	4.07%	1.48%	0.59%
DroidLysis	78.78%	1.16%	11.85%	1.22%	86.58%	0.96%	14.71%	0.65%

Table 6.6: Number of True-Positives and False-Positives for each detector evaluated on the different datasets. We found DroidLysis to work unreliable, giving different results for each run.
The best results per column are marked green.

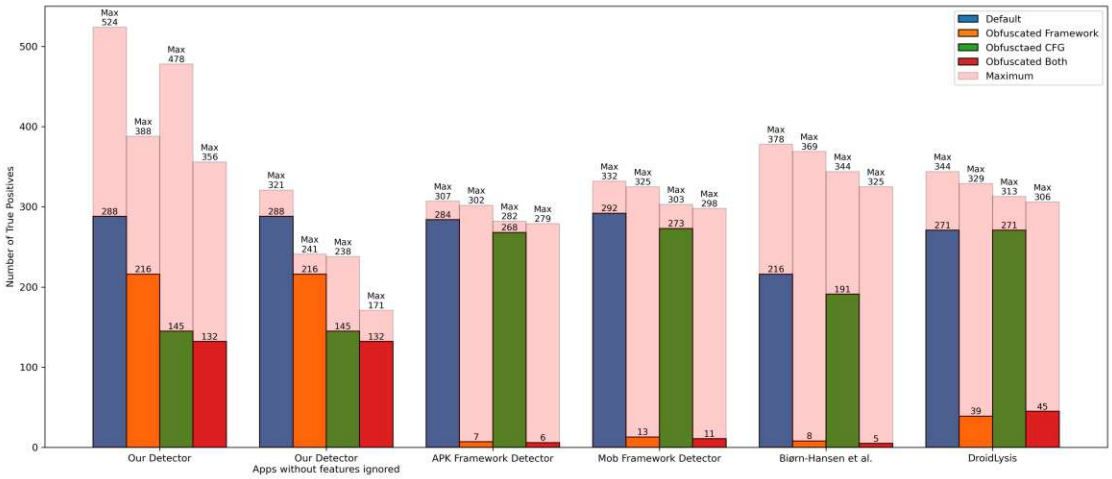

Figure 6.1: Number of applications, where the framework could be correctly identified grouped by the different detectors. The red bar indicates the maximum number of apps for which the framework could have been detected by each detector on the given dataset.

6.2.4 Summary of the results

In summary, we can conclude that the function features are suitable for detecting the framework used in an application, as shown by the results for our detector when ignoring applications without features. However, the extraction of these function features has been shown to be unreliable, hindering the performance of our detector. Therefore, it only makes sense to be applied over existing detectors, such as the APK Framework Detector or the Mob Framework Radar, if these do not return possible frameworks due

to an application being obfuscated. This is due to the simplicity of using directory, file and class names for detection giving solid results whilst being fast and easy to check. The existing detectors, except for DroidLysis, took less than 5 minutes for the whole dataset, most of this time stems from overhead of running a Docker container for each application as well as having the APK files stored on a remote system mounted via *sshfs*[1]. DroidLysis took around 50 minutes to analyze the whole dataset, due to it performing other more time-consuming analysis at the same time, which were not necessary for the detection of frameworks, but could not be turned off without modification. However, this is still faster than the time needed for running our detector, whilst the comparison of the applications' functions with the relevant functions for each framework is rather fast and takes no more than 15 minutes for the whole dataset, the disassembly and extraction of the functions and their features using *TikNib* and *IDA Pro* takes most of the time around 30 minutes per application. Therefore, the bottleneck of our detector both for reliability and the time needed for analysis, is the decompilation and extraction of the function features. Future work should therefore focus on improving the extraction of the function features. Furthermore, we have seen that even simple obfuscations applied just using bash scripts is enough to significantly reduce the performance of the existing framework detectors. Using obfuscation techniques that modify the applications' CFG, CG and instructions have an impact on the performance of our detector, but it still manages to produce usable results.

6.3 Threats to Validity

Firstly, certain frameworks, such as Apache Cordova, Apache Flex, Capacitor, Kivy, Titanium SDK, Unity, etc., were under-represented in our dataset, primarily because information about the frameworks used by an application is often not publicly available. For the creation of our dataset we therefore relied on application showcases on the frameworks' websites and information in the metadata files for applications from the F-Droid store. As a result, the evaluation results for under-represented apps might be unreliable. Further evaluation should therefore try to increase the number of applications for these frameworks. In addition, the small number of applications hurts the identification of relevant functions used by our detector. Frameworks with a bigger representation in our dataset were detected more reliable.
Due to the small number of applications for some frameworks, we did not split our dataset in a training and evaluation set, as this would have further reduced the number of applications available for evaluation and training. As a consequence, our detector could perform different if evaluated on a dataset not used for identifying the relevant functions used for detection. As we limited the number of relevant functions used for detection per framework to 30, we argue that such an over fitting to our dataset is unlikely, at least for frameworks not under-represented.

[1] `https://github.com/libfuse/sshfs`

Our dataset contained a different number of applications for each framework, with some frameworks, like Flutter, being overrepresented in our dataset. This can base the evaluation results towards detectors that perform good on these overrepresented frameworks.
After obfuscating the applications in our dataset we did not check whether they could still be installed and run. We checked that the obfuscation scripts developed in chapter 4 produced runnable applications when applied to the apps used to developed them, hence we are confident that this also applies to the majority of the apps in our dataset.
Obfuscapk, used in the creation of the *Obfuscate-CFG* and *Obfuscate-Both* datasets, has been archived on the 15th November 2023, we used it for obfuscation in January 2024, hence we consider it still useable for obfuscating APK files. In addition, as we have seen in chapter 4, Obfuscapk may not always apply as much obfuscation as possible or expected. As a result the apps obfuscated using Obfuscapk might not be comparable to apps obfuscated using more sophisticated obfuscators. Furthermore, most obfuscators are part of the build chain and not applied to the APK like Obfuscapk. As these have access to the source code and are applied during compilation they can perform more sophisticated and precise obfuscation. It is therefore possible that the detectors evaluated perform different when run on applications obfuscated during compilation.
We refrained from comparing the run-time of each detector in more detail, as the overhead caused by the evaluation setup, running them in Docker containers and the file system holding the APK files introduced to many variables.
During evaluation, we have seen DroidLysis perform differently between runs, due to it sometimes crashing running into timeouts. This non-deterministic behavior makes precise comparisons of DroidLysis challenging.

CHAPTER 7

Conclusion and Future Work

7.1 Summary

In this thesis we looked into the functioning of current framework detection techniques and showed them in chapter 4 to easily be circumvented by simple file, directory and class renaming performed using standard command line tools and bash scripts. Motivated by this we developed a novel approach to framework detection utilizing function features extracted from the application's binary OAT representation that are resilient against this kind of obfuscation. We showed the feasibility of function features for framework detection, but also discovered their extraction to be unreliable and time-consuming.
We will now review the research questions from section 1.3.

R1 - How do the current framework detection tools perform compared to each other?

We have seen the current framework detectors to perform well on the default dataset, with the APK Framework Detector having the highest true-positive percentage. Compared to each other their performance was similar except for the detector by Biørn-Hansen et al. [BHGM+22], which performed significantly worse. This is rather surprising as it represents the latest research paper developing a framework detector, whilst at the same time utilizing the same techniques as the other detectors. Its poor performance can thus be attributed to the poor selection of the artifacts chosen for identifying the frameworks. The current detectors however, support detecting only a limited number of frameworks each, limiting the application.

R2 - How does obfuscation affect the performance of the different framework detection tools?

Obfuscation aimed at the applications' code, changing its functions CG and CFG had no impact on the performance of the current framework detectors, as these did not rely on features modified by these obfuscations. However, applying the obfuscations developed in chapter 4 strongly reduced the performance of the detectors. As we have seen that these kinds of obfuscations require handling framework specific special cases in order to produce functioning applications the real world impact of these on framework detection has to be seen.

R3 - How does our novel approach perform compared to the tools tested in **R1** and **R2**?

Our detector performs worse on the default dataset than the current techniques. This is in first due to the high rate of applications where no function features could be extracted. When ignoring apps without function features we can see our detector to have a similar true-positive percentage as the existing detectors, be it with a slightly higher false-positive percentage. When taking obfuscation into consideration we can see that obfuscation aimed against the applications' code decreases the performance of our detector, whilst obfuscation of file, directory, class and library names has little to no effect. With the latter reducing the effectiveness of the current framework detectors heavily, hence we deem our detector the only one being robust against this kind of obfuscation.

R4 - Is binary code analysis of an Android applications OAT representation a feasible approach for detecting its used frameworks?

We deem usage of the OAT representation for binary code analysis a feasible approach for framework detection, due to its comparable results and increased robustness against obfuscation compared to current framework detectors. However, time and computational efforts required for the binary code analysis is higher than for the current detectors. As creation of the OAT files itself can be done in little time, future research should empathize on decreasing the time needed for extraction of the function features, as well as the analysis of the OAT files, in order to decrease the time and effort needed for applying this approach.

7.2 Future Work

We propose the following tasks to follow our research.

- We identified the extraction of the function features as the main obstacle, due it being time-consuming and not very reliable. Therefore, future research should look into alternative and faster ways for extracting these from the applications' OAT representation. One possible option could be to take them from the *oatdump* generated by the OAT compiler. The *oatdump* contains the Dalvik code side by side with the binary code it is replaced with for each function. Information present in there could significantly speed up the extraction of the function features.

- As some frameworks are under-represented in our dataset, increasing the number of apps for these frameworks can lead to better and more significant evaluation results. At the same time it can aid the identification of relevant functions for these frameworks. Future research should therefore try to identify more applications using these frameworks, further increasing the dataset size.
- The different framework detectors can be applied to applications using unknown or no frameworks and their results compared. This could be the basis for the creation of a larger dataset of applications and the frameworks they are using, whilst at the same time giving an overview about usage of different frameworks the wild.
- The current scoring method used by our detector is simple, future work could look into more sophisticated ways of scoring the function features. At the same time the impact of the different kind of function features could be investigated.
- Most obfuscators are part of the application's build chain and applied during the compilation. However, the obfuscators used in this thesis were applied after the compilation on the APK files. Evaluating the different detectors on this kind of obfuscators can give further insights into the obfuscation resilience of the different framework detectors. It should also be investigated how they handle framework specific special cases.
- As many frameworks use their own libraries, identification of these could aid in the detection of frameworks. None of the current framework detectors use library files apart from their file names for detection. As seen in chapter 3 library detection is studied extensively, therefore it should be looked into whether such techniques can be used for framework detection.

Acronyms

AAB Android App Bundle. 3, 5, 6, 8

AAPT2 Android Asset Packaging Tool. 7

AOT Ahead-of-Time. 8

APK Android Package. 3, 5–9, 11, 14, 17–19, 25–30, 32, 33, 36, 41–43, 47–51, 53

ART Android Runtime. 7, 8

AST Abstract Syntax Tree. 14

CFG Control Flow Graph. 14, 15, 22, 36, 38, 40, 43, 49, 52

CG Call Graph. 14, 15, 22, 36, 38, 43, 49, 52

CNN Convolutional Neural Network. 22

CPDF Cross-Platform Development Framework. xi, xiii, 1, 3, 8, 9, 11, 17, 18, 32

DEX Dalvik Executable. 6, 13

JIT Just-in-Time. 7

JNI Java Native Interface. 7, 29, 30

JSI JavaScript Interface. 11

JVM Java Virtual Machine. 7

MPNN Message Passing Neural Network. 22

NLP Natural Language Processing. 22

OAT Of Ahead Time. xi, xiii, 1, 2, 8, 36, 38, 40, 41, 51, 52

TPL Third-Party Library. 19, 21, 22

Bibliography

[ABKLT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. Association for Computing Machinery.

[AM16] Mohamed Ali and Ali Mesbah. Mining and characterizing hybrid apps. In *Proceedings of the International Workshop on App Market Analytics*, WAMA 2016, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[Anda] AAPT2, Available at: `https://developer.android.com/tools/aapt2` (Accessed: 2023-09-01).

[Andb] About Android App Bundles, Available at: `https://developer.android.com/guide/app-bundle` (Accessed: 2023-08-31).

[Andc] Android Multidex, Available at: `https://web.archive.org/web/20231214153927/https://developer.android.com/build/multidex` (Accessed: 2023-12-14).

[Andd] App manifest overview, Available at: `https://developer.android.com/guide/topics/manifest/manifest-intro` (Accessed: 2023-09-01).

[Ande] App resources overview, Available at: `https://developer.android.com/guide/topics/resources/providing-resources` (Accessed: 2023-09-01).

[Andf] Application Fundamentals, Available at: `https://developer.android.com/guide/components/fundamentals` (Accessed: 2023-09-01).

[Andg] Build your app from the command line, Available at: `https://developer.android.com/build/building-cmdline` (Accessed: 2023-08-31).

[Andh] Configure your build, Available at: `https://developer.android.com/build` (Accessed: 2023-08-31).

[Andi] d8, Available at: https://developer.android.com/tools/d8 (Accessed: 2023-09-01).

[Andj] Get started with the NDK, Available at: https://developer.android.com/ndk/guides (Accessed: 2023-09-01).

[Andk] Reduce your app size, Available at: https://developer.android.com/topic/performance/reduce-apk-size (Accessed: 2023-09-01).

[Andl] Shrink, obfuscate, and optimize your app, Available at: https://developer.android.com/build/shrink-code (Accessed: 2023-09-01).

[Ang] AngularJS, Available at: https://angularjs.org/ (Accessed: 2023-10-19).

[Apaa] Apache Cordova, Available at: https://cordova.apache.org/ (Accessed: 2023-09-05).

[Apab] Apache Flex, Available at: https://flex.apache.org/index.html (Accessed: 2023-10-16).

[APKa] APK Framework Detector - Github, Available at: https://github.com/EngineerDanny/apk-framework-detector (Accessed: 2023-08-29).

[APKb] APK Platform Detector, Available at: https://play.google.com/store/apps/details?id=com.alidaaer.platformdetector (Accessed: 2023-08-31).

[BBD16] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 356–367, New York, NY, USA, 2016. Association for Computing Machinery.

[BBM+18] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[BHGG18] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Comput. Surv.*, 51(5), nov 2018.

[BHGG+19] Andreas Biørn-Hansen, Tor-Morten Grønli, Gheorghita Ghinea, Sahel Alouneh, et al. An empirical study of cross-platform mobile development in industry. *Wireless Communications and Mobile Computing*, 2019, 2019.

[BHGM+22] Andreas Biørn-Hansen, Tor-Morten Grønli, Tim Alexander Majchrzak, Hermann Kaindl, and Ghinea Gheorghita. The use of cross-platform frameworks for google play store apps. 2022.

[BL23] Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART, Available at: https://doi.org/10.1145/3578357.3591219.

[Cap] Capacitor, Available at: https://capacitorjs.com/ (Accessed: 2023-10-16).

[CGO15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[Coo] How the React Native bridge works and how it will change in the near future, Available at: https://dev.to/wjimmycook/how-the-react-native-bridge-works-and-how-it-will-change-in-the-near-future-4ekc (Accessed: 2023-09-05).

[Das] DashO | PreEmptive, Available at: https://www.preemptive.com/products/dasho/ (Accessed: 2023-09-06).

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[Dexa] Android App Security and Obfuscation | DexGuard, Available at: https://www.guardsquare.com/dexguard (Accessed: 2023-09-06).

[Dexb] DexProtector | Licel, Available at: https://licelus.com/products/dexprotector (Accessed: 2023-09-06).

[DLD+18] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[Dro] DroidLysis - Github, Available at: https://github.com/cryptax/droidlysis/tree/master (Accessed: 2023-11-09).

[Exo] Exodus Privacy, Available at: https://reports.exodus-privacy.eu.org/en/trackers/stats/ (Accessed: 2023-09-14).

[Exp] Expo, Available at: https://expo.dev/ (Accessed: 2023-10-16).

[Flu] Flutter, Available at: https://flutter.dev/ (Accessed: 2023-09-05).

[Fra] Framework7, Available at: https://framework7.io/ (Accessed: 2023-10-16).

[GSR+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[HC21] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), apr 2021.

[Iona] Ionic, Available at: https://ionicframework.com/ (Accessed: 2023-10-16).

[Ionb] Ionic Documentation, Available at: https://ionicframework.com/docs (Accessed: 2023-10-19).

[Kiv] Kivy, Available at: https://kivy.org/ (Accessed: 2023-10-17).

[KKC+23] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2023.

[LB22] Martina Lindorfer and Jakob Bleier. Mobile security (android). Presentation slides, 2022. Presented as part of the System & Application Security course at the Vienna University of Technology.

[LBK21] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 47(4):676–693, April 2021.

[Lev] Android CPU, Compilers, D8 & R8, Available at: https://proandroiddev.com/android-cpu-compilers-d8-r8-a3aa2bfbc109?gi=5ca82e9e790a (Accessed: 2023-09-04).

[LHZ+18] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 667–678, New York, NY, USA, 2018. Association for Computing Machinery.

[Liba] LibPecker - Github, Available at: https://github.com/yuanxzhang/LibPecker (Accessed: 2023-08-29).

[Libb] LibRadar - Github, Available at: https://github.com/pkumza/LibRadar (Accessed: 2023-07-03).

[Libc] LibScout - Github, Available at: https://github.com/reddr/LibScout (Accessed: 2023-07-03).

[lie] Android formats — LIEF Documentation, Available at: https://lief-project.github.io/doc/latest/tutorials/10_android_formats.html (Accessed: 2023-09-11).

[LK09] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[LLB+17] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[LLJG15] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 89–103, New York, NY, USA, 2015. Association for Computing Machinery.

[MA21] Mohammad Mahendra and Bayu Anggorojati. Evaluating the performance of android based cross-platform app development frameworks. In *Proceedings of the 6th International Conference on Communication and Information Processing*, ICCIP '20, page 32–37, New York, NY, USA, 2021. Association for Computing Machinery.

[MD2] MD² - Model-driven Mobile Development, Available at: https://wwu-pi.github.io/md2-web/ (Accessed: 2023-10-19).

[MDC+21] Sergio Mascetti, Mattia Ducci, Niccolò Cantù, Paolo Pecis, and Dragan Ahmetovic. Developing accessible mobile applications with cross-platform development frameworks. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[MdTGM19] O. Mirzaei, J.M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[Med] The internals of Android APK build process — Article, Available at: https://medium.com/androiddevnotes/the-internals-of-android-apk-build-process-article-5b68c385fb20 (Accessed: 2023-09-04).

[Met] Meteor, Available at: https://www.meteor.com/ (Accessed: 2023-10-18).

[Mob] Mob Framework Radar - Github, Available at: https://github.com/papocch10/mob-framework-radar (Accessed: 2023-08-29).

[MRST15] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 56–59, 2015.

[MS21] Abhinav Mohanty and Meera Sridhar. Hybridiagnostics: Evaluating security issues in hybrid smarthome companion apps. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 228–234, 2021.

[MWGC16] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, page 653–656, 2016.

[Nat] NativeScript, Available at: https://nativescript.org/ (Accessed: 2023-10-16).

[Now] Think Twice Before Adopting Security By Obscurity in Kotlin Android Apps, Available at: https://www.nowsecure.com/blog/2019/07/11/think-twice-before-adopting-security-by-obscurity-in-kotlin-android-apps/ (Accessed: 2023-08-31).

[Pat] Deep dive into React Native's New Architecture, Available at: https://medium.com/coox-tech/deep-dive-into-react-natives-new-architecture-fb67ae615ccd (Accessed: 2023-09-05).

[Phoa] Goodbye PhoneGap, Available at: https://cordova.apache.org/announcements/2020/08/14/goodbye-phonegap.html (Accessed: 2023-10-18).

[Phob] PhoneGap, Cordova, and what's in a name?, Available at: https://web.archive.org/web/20131201015613/http://phonegap.com/2012/03/19/phonegap-cordova-and-what\T1\textquoterights-in-a-name/ (Accessed: 2023-09-05).

[Phoc] PhoneGap End of Service, Available at: https://helpx.adobe.com/de/experience-manager/kb/adobe-phonegap-end-of-service.html (Accessed: 2023-10-18).

[Pow] Power APK - Google Play Store, Available at: https://play.google.com/store/apps/details?id=eu.sisik.apktools (Accessed: 2023-08-31).

[QtF] Qt Framework, Available at: https://www.qt.io/product/framework (Accessed: 2023-10-19).

[Reaa] React, Available at: https://de.legacy.reactjs.org/ (Accessed: 2023-10-19).

[Reab] React Native, Available at: https://reactnative.dev/ (Accessed: 2023-09-05).

[Reac] React Native - Architecture Overview, Available at: https://reactnative.dev/architecture/overview (Accessed: 2023-09-05).

[Read] Recat Native - Why a New Architecture, Available at: https://reactnative.dev/docs/the-new-architecture/why (Accessed: 2023-09-05).

[Reae] Turbo Native Modules, Available at: https://reactnative.dev/docs/the-new-architecture/pillars-turbomodules (Accessed: 2023-09-04).

[Rei91] Christoph Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 144–152, 1991.

[Rho] RhoMobile, Available at: https://tau-platform.com/en/products/rhomobile/ (Accessed: 2023-10-16).

[Sis] Sisik Power APK, Available at: https://www.sisik.eu/blog/android/apk/analyze-app-to-find-framework (Accessed: 2023-08-29).

[Sol] Solar2D, Available at: https://solar2d.com/ (Accessed: 2023-10-16).

[Staa] Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2022, Available at: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (Accessed: 2023-09-22).

[Stab] Number of apps available in leading app stores as of 3rd quarter 2022, Available at: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (Accessed: 2023-09-01).

[Tit] Titanium SDK, Available at: https://titaniumsdk.com/ (Accessed: 2023-10-16).

[TJM+21] Donghai Tian, Xiaoqi Jia, Rui Ma, Shuke Liu, Wenjing Liu, and Changzhen Hu. Bindeep: A deep learning approach to binary code similarity detection. *Expert Systems with Applications*, 168:114348, 2021.

[Uni] Unity Mobile, Available at: https://unity.com/solutions/mobile (Accessed: 2023-09-05).

[Uno] Uno Platform, Available at: https://platform.uno/ (Accessed: 2023-10-16).

[VGN14] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, jun 2014.

[WHA+18] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 222–235, New York, NY, USA, 2018. Association for Computing Machinery.

[WWZR18] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-resilient library detection for android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, page 13–23, New York, NY, USA, 2018. Association for Computing Machinery.

[Xam] Xamarin, Available at: https://dotnet.microsoft.com/en-us/apps/xamarin (Accessed: 2023-10-02).

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, page 213–220, New York, NY, USA, 2013. Association for Computing Machinery.

[YCT+20] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1145–1152, 2020.

[ZBLO21] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.

[ZDZ+18] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152, 2018.

[ZFL+20] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for android applications: Are we there yet? In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 919–930, 2020.

[ZLF+22] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering*, 48(10):4181–4213, 2022.